VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERAKTIVNÍ SIMULÁTOR PRO GRAFY TOKU DAT

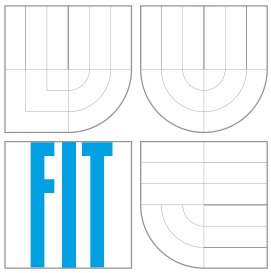
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID KOVAŘÍK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERAKTIVNÍ SIMULÁTOR PRO GRAFY TOKU DAT

AN INTERACTIVE SIMULATOR FOR DATA-FLOW GRAPHS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID KOVAŘÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ CHARVÁT

BRNO 2015

Abstrakt

Grafy toku dat jsou často používány při návrhu hardware. Jsou však vhodné také pro provádění hlubších analýz návrhů (např. funkční a formální verifikace). Simulátor prezentovaný v této práci vzniká jako podpůrný nástroj pro verifikační prostředí HADES. Cílem simulátoru je snížit potřebný čas a zvýšit kvalitu procesu verifikace. Pro efektivní provádění simulace byl navržen a implementován simulační algoritmus, který díky eliminaci nadbytečných vyhodnocení šetří výpočetní čas. Simulátor je vybaven několika výstupními rozhraními, která jsou připojena k simulačnímu jádru. První rozhraní poskytuje přímý výstup simulace v textové podobě. K němu existuje také interaktivní varianta, která dovoluje uživateli řídit běh simulace a manipulovat se stavem modelu. Třetí vytváří plnohodnotné uživatelské rozhraní určené pro vizualizaci průběhu simulace.

Abstract

Data-flow graphs are often used by hardware designers. Such graph representation is also very useful for performing deeper analysis of a design (including functional or formal verification). Simulator presented in this thesis is a support tool for verification environment HADES. The goal of the simulator is to decrease necessary time and increase quality of the verification process. To perform a simulation efficiently, a specific simulation algorithm which saves computation time by eliminating redundant evaluations has been introduced. The simulator is equiped with several output interfaces connected to a single simulation core. One output interface provides direct simulation output in text format. The second is also textual, but allows user to control the simulation. Finally, the third forms a graphical interface that visualizes simulation results.

Klíčová slova

modelování, simulace, grafy toku dat, testování, vizualizace

Keywords

modeling, simulation, data-flow graphs, testing, Visualisation

Citace

David Kovařík: Interaktivní simulátor pro grafy toku dat, bakalářská práce, Brno, FIT VUT v Brně, 2015

Interaktivní simulátor pro grafy toku dat

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Charváta.

.....
David Kovařík
18. května 2015

Poděkování

Děkuji Ing. Lukáši Charvátovi za poskytnutí odborné pomoci a rad jak při návrhu a implementaci výsledného programu, tak při psaní textu této práce.

Obsah

Kapitola 1

Úvod

Vzhledem k vysokému stupni paralelismu, který se v hardware nachází, nejsou pro jeho popis vhodné klasické sekvenční jazyky. Proto vznikly jazyky specializované, jejichž konstrukce dovolují paralelismus snadno modelovat. Jedná se například o jazyky VHDL či Verilog. Pro potřeby hlubšího zkoumání vlastností modelovaného hardware je často vhodné transformovat jeho popis do grafové reprezentace. Mezi návrháři jsou velmi oblíbené *grafy toku dat*, především pro jejich schopnost implicitně modelovat vysoký stupeň paralelismu.

Cílem této práce je vytvořit interaktivní simulátor grafů toku dat. Simulátor `dfsim` bude podpůrným nástrojem pro verifikační prostředí *HADES* [13, 12], které slouží k detekci možných datových hazardů v modelu procesoru. Úkolem simulátoru je snížit potřebný čas a zvýšit kvalitu celého procesu verifikace. Důležitým požadavkem je také efektivita provádění simulace. Z tohoto důvodu byl navržen a implementován specifický simulační algoritmus, který eliminuje nadbytečná vyhodnocení uzlů grafu a šetří tak výpočetní čas.

Jelikož se jedná o podpůrný nástroj, poskytuje simulátor uživateli několik různých přístupů k analýze běhu simulace. Pro jednoduché ovládání v příkazové řádce má uživatel k dispozici textové rozhraní, které poskytuje přímý výstup simulace. K němu existuje také interaktivní varianta, která uživateli dovoluje zadávat příkazy a řídit tak běh simulace nebo manipulovat se stavem modelu. Simulátor také obsahuje plnohodnotné grafické rozhraní, jehož úkolem je vizualizovat průběh simulace, aby byl uživatel schopen snadněji pochopit chování modelu.

Kapitola 2 popisuje existující modely, technologie a nástroje, které se při popisu hardware používají. Úvodní část kapitoly (sekce 2.1) se zaměřuje na existující jazyky pro popis hardware. Cílem není popsat jejich syntaxi či implementační detaily, ale shrnout jejich silné a slabé stránky a principy, které se v nich využívají. Sekce 2.2 je zaměřena na popis nejčastěji používaných grafových modelů, které jsou při popisu hardware používány. Jako poslední jsou v této kapitole, v sekci 2.3, zmíněny existující simulátory hardware.

Kapitola 3 je věnována návrhu jednotlivých komponent simulátoru a jejich interakce mezi sebou. Nejdříve jsou v sekci 3.1 zmíněny jednotlivé požadavky, které byly na návrh a implementaci kladeny. V sekci 3.2 je podrobně popsána syntaxe a sémantika vstupního jazyka. Sekce 3.3 a 3.4 popisují návrh simulačního jádra, propojení jednotlivých komponent a interakci s uživatelem. Poslední sekce této kapitoly (sekce 3.6) představuje jednotlivé režimy výstupu simulace.

Provedené simulační experimenty a testovací sada simulátoru jsou demonstrovány v kapitole 5. Nejdříve jsou zmíněny jednodušší modely a ukázány výsledky jejich simulace. Postupně se pak přechází k experimentům se složitějšími modely jako např. model 8-bitového procesoru.

Kapitola 2

Modely pro popis hardware a jejich simulace

Tato kapitola je zaměřena na popis jednotlivých jazyků pro popis hardware, grafových modelů pro jeho modelování a také jejich existující simulátory. Cílem je shrnout klady a zápory nástrojů, které jsou v dnešní době na poli modelování hardware k dispozici v různých fázích vývoje.

2.1 Jazyky pro popis hardware

Zásadní rozdíl mezi hardware a software je míra paralelismu, který se v nich nachází. Software je typicky sekvenční program, proto pro jeho popis dostačují sekvenční programovací jazyky jako např. jazyk C nebo C++. Proti tomu stojí hardware, který je charakteristický svým vysokým stupněm paralelismu. Tento rozdíl činí jazyky pro popis software nevhodné pro popis hardware (a naopak). Z toho důvodu vznikla řada jazyků, které se soustředí právě na popis hardware. Příkladem mohou být jazyky *VHDL* (sekce 2.1.1) a *Verilog* (sekce 2.1.2).

2.1.1 VHDL

Jazyk VHDL [18] patří do skupiny jazyků pro popis hardware (*angl., hardware description language – HDL*). Je určen pro popis číslicových systémů (např. integrovaných obvodů či hradlových polí). Popisuje číslicová zařízení a jejich části pomocí *entit* a *architektur*. Entita specifikuje rozhraní komponenty, zatímco architektura popisuje její chování a funkci.

Architektura je vždy svázána s entitou, která definuje patřičné rozhraní, a může být popsána na úrovni *struktury*, *chování (behaviorální popis)* nebo *toku dat*. Při behaviorálním popisu je popis architektur složen z jednoho či více procesů. Cílem je popsat, jakým způsobem se mění výstupní hodnoty v závislosti na vstupu. Popis toku dat modeluje datové závislosti mezi entitami. Strukturální přístup popisuje, z čeho se daný systém skládá (tj. jakou má strukturu). Navíc dovoluje hierarchické skládání jednotlivých struktur.

2.1.2 Verilog

Jazyk Verilog [22] také patří do rodiny jazyků HDL. Používá se pro návrh a verifikaci číslicových obvodů na úrovni přenosů dat mezi registry. Lze jej použít také pro popis analogových systémů.

Jazyk je volně inspirovaný jazykem C, který byl při vývoji Verilogu již zaveden a používán pro vývoj software. Podobně jako jazyk C obsahuje Verilog jednoduchý textový preprocesor. Shodná s jazykem C jsou také klíčová slova pro řízení toku programu a jejich precedence.

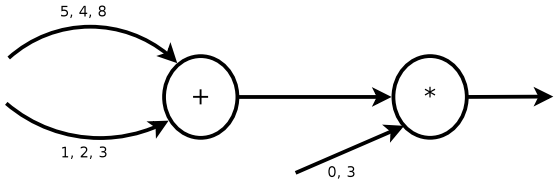
Návrh ve Verilogu představuje hierarchii modulů, které se skládají z deklarací proměnných, konkurentních a sekvenčních bloků a instancí jiných modulů. Jelikož jednotlivé moduly běží konkurentně, je jazyk Verilog označován za jazyk pro popis toku dat.

2.2 Grafy toku dat

Grafy toku dat jsou schopny popsat vysoký stupeň paralelismu a asynchronní chování systému [16], proto se jejich použití rozšířilo například na poli zpracování digitální signálů. Komplexní výpočetní úloha je v grafu rozdělena na menší podúlohy, které jsou mezi sebou propojeny tak, aby si mohly předávat výsledky.

2.2.1 Neformální popis grafů toku dat

Graf toku dat [11] je orientovaný graf skládající se z konečného počtu vrcholů, které jsou mezi sebou propojeny hranami. Každý vrchol modeluje výpočetní funkci (transformuje vstupy na výstup), zatímco každá hrana představuje datovou cestu mezi dvěma uzly. Uzly mezi sebou mohou komunikovat pouze pokud mezi nimi existuje datová cesta. V obecné definici každá datová cesta představuje FIFO frontu, ve které se postupně řadí tzv. *tokeny* (jednotky dat).



Obrázek 2.1: Jednoduchý graf toku dat.

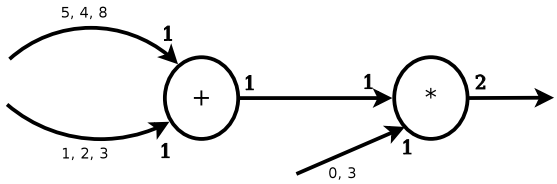
Na obrázku 2.1 je vidět jednoduchý graf toku dat. Sémantika modelu je taková, že každý uzel může spustit výpočet nové výstupní hodnoty kdykoli jsou připravena potřebná data na vstupních hranách. Uzel bez vstupních hran může být aktivován kdykoliv. Proto jsou grafy toku dat označovány jako *řízené daty* (*angl., data–driven*). Díky této vlastnosti není potřeba centrální prvek systému, který by výpočet řídil.

Výpočetní operace jednotlivých uzlů nesmí mít žádný vedlejší efekt (čtení/zápis globálních proměnných atp. . .). Pokud by toto pravidlo bylo porušeno, nebylo by možné zajistit integritu modelu. Komunikace mezi uzly je možná výhradně přes datové cesty mezi nimi.

Obecně jsou grafy toku dat *bezčasové*. Aktivace funkčních uzlů a vyhodnocení jejich funkce probíhá v nulovém čase (přestože při reálné implementaci je pro výpočet výstupní hodnoty potřeba nenulový čas). Bezčasový v tomto kontextu znamená to, že čas je nevýznamný z pohledu plánování aktivace uzlů. O aktivaci uzlu je rozhodnuto na základě přítomnosti vstupních dat. Funkční uzel není nikdy aktivován, pokud na jeho vstupu není dostatečné množství tokenů [20].

V grafech je také možné modelovat tzv. *zpoždění*, která představují odstup tokenů mezi vstupem a výstupem (nikoli zpoždění z hlediska času). Jsou modelována jako počáteční

tokeny ve frontách datových cest. Např. na obrázku 2.1 by jednotkové zpoždění mezi uzlem + a ∗ znamenalo, že n-tý token konzumovaný uzlem ∗ bude token číslo $(n-1)$ produkovaný uzlem +. Zpoždění jsou vyžadována při modelování zpětných vazeb. Bez počátečních tokenů na zpětnovazební hraně by funkční uzel, k němuž je tato hrana připojena na vstup, nikdy nebyl aktivován, protože by se na této hraně nikdy neobjevilo dostatečné množství tokenů.



Obrázek 2.2: Graf toku dat s ohodnocenými hranami.

Jednotlivé hrany mohou být ohodnoceny *váhami* (lze vidět na obrázku 2.2). Váhy udávají počet potřebných tokenů na vstupních hranách pro aktivaci uzlu a počet produkovaných tokenů výstup. Existují různá pravidla pro konzumaci a produkci tokenů, na jejichž základě jsou popsány různé typy grafů toku dat. Mezi návrháři hardware jsou nejčastěji používány takové grafy, v nichž je počet konzumovaných a produkovaných tokenů znám předem pro každý uzel a během existence grafu se již nemění. Tyto grafy jsou označovány jako *synchronní grafy toku dat* (*angl., synchronous data-flow graphs – SDF*). Jejich vlastnosti jsou více popsány v sekci 2.2.2. Grafy toku dat, v nichž toto pravidlo neplatí, jsou označovány jako *asynchronní*.

Jelikož v grafu může dojít k souběžné aktivaci více uzlů, je při reálné aplikaci nutno zavést tzv. *evaluační plán*, který určuje, kdy je který uzel aktivován – kdy jsou mu přiřazeny reálné zdroje.

2.2.2 Synchronní grafy toku dat

Jako *synchronní* jsou označovány takové grafy toku dat, ve kterých je počet konzumovaných/produkovaných tokenů konstantní a specifikován pro každý uzel předem [20]. Struktura grafu a poměry konzumovaných/produkovaných tokenů rozhodují o tom, zda je možné pro graf sestavit *přípustný evaluační plán*. Jedná se o takový plán, na jehož základě lze vyhodnocovat uzly grafu nekonečně dlouho, aniž by došlo k uváznutí nebo nekonečnému narůstání front datových cest. Sestavování plánu není předmětem této práce, více informací o dané problematice lze nalézt v [16, 20].

Oproti asynchronním grafům toku dat mají SDF grafy zásadní výhodu. Potřebné informace pro sestavení plánu jsou známy předem, je tedy možné plán sestavit ještě před spuštěním. Plán se pak již nemění, což výrazně snižuje režii v době běhu. Nicméně SDF grafy vykazují horší vlastnosti v oblastech souvisejících s řízením toku dat, například:

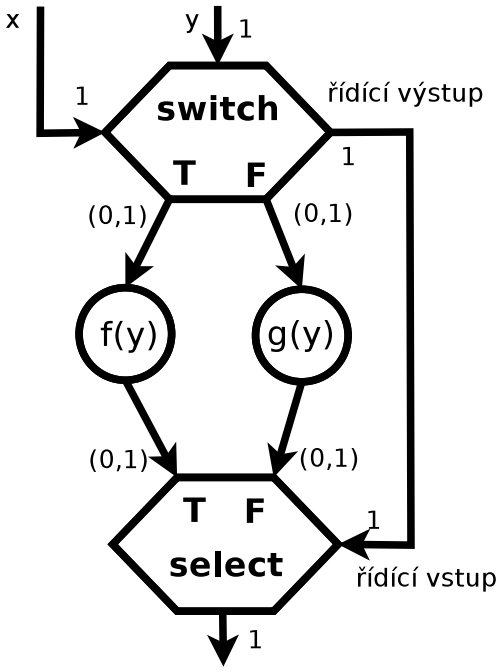
- **zastavení a restartování modelu** – je problematické zajistit zastavení běhu modelu nebo jej uvést do počátečního stavu,
- **přepínání režimů** – topologie grafu je fixní a nelze ji za běhu měnit či přepínat,
- **výjimky** – jakmile token vstoupí do fronty na datové cestě, jediná možnost, jak může frontu opustit je, že jej z ní vytáhne příslušný funkční uzel,

- **podmínky za běhu** – jednoduchá konstrukce `if-then-else` je pro SDF grafy také problémová (nelze určit, kdy má být jaký token odeslán na kterou výstupní hranu).

2.2.3 Asynchronní grafy toku dat

Asynchronní jsou takové grafy toku dat, u nichž nelze předem určit počet konzumovaných/produkovaných tokenů při aktivaci jednotlivých uzlů. Nelze pro ně předem sestavit proveditelný plán pro mapování uzlů na reálné zdroje [16], což s sebou nese vyšší režii za běhu. Na druhou stranu umožňují pohodlněji modelovat řídící konstrukce (např. `if-then-else`).

Asynchronní může být buď celý graf, nebo jen některé jeho uzly. Na obrázku 2.3 je znázorněn graf toku dat, který modeluje konstrukci `if(x) then f(y) else g(y)`. Tento graf obsahuje dva asynchronní uzly (`switch` a `select`). Výstupní váhy uzlu `switch` jsou definovány jako dvojice `(0, 1)`, což znamená, že při aktivaci uzlu je na výstupní hranu produkován žádný, nebo jeden token. Jejich počet je určen na základě hodnoty vstupního signálu `x`. Množství tokenů na výstupu tedy nelze určit předem.



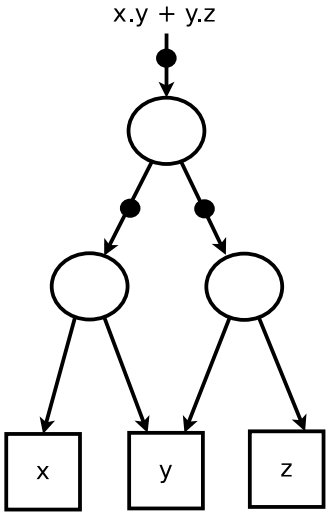
Obrázek 2.3: Asynchronní graf toku dat pro konstrukci `if-then-else`. Převzato z [16].

2.2.4 And-invertor grafy (AIG)

Přestože tento typ grafů nebývá přímo označován jako graf toku dat, slouží také pro popis transformace vstupů na výstupy (podobně jako grafy toku dat). AIG je orientovaný acyklický graf, který popisuje funkci logických obvodů. Nalezly užití v různých nástrojích pro automatizaci návrhu elektronických obvodů nebo v oblasti formální verifikace. [23]

AIG se skládá z dvouvstupých hradel `AND`, invertorů, listových uzlů představujících proměnné a hran, které je propojují. Některé definice AIG zavádí invertory jako samostatné uzly grafu, v jiných definicích jsou modelovány jako speciální typ hran. Pro transformaci

logických funkcí do AIG popisu stačí vyjádřit funkci výhradně pomocí operací logického součinu a inverze. Ty pak mohou být reprezentovány pomocí hradel `AND` a invertorů. Příklad AIG grafu je ukázán na obrázku 2.4.



Obrázek 2.4: Ukázka jednoduchého grafu AIG.

2.3 Existující simulátory

V dnešní době existuje celá řada simulátorů hardware nebo abstraktních modelů pro jejich popis (např. grafy). Každý z nich má své charakteristické vlastnosti. Tato sekce se věnuje stručnému popisu vybraných z nich.

2.3.1 ModelSim

ModelSim [6] je komerční nástroj pro simulaci a verifikaci návrhů systémů zapsaných např. v jazycích VHDL, Verilog, SystemVerilog. Je vhodný pro simulaci jak aplikačně specifických obvodů, tak FPGA polí. Mezi nejvýraznější vlastnosti tohoto nástroje patří:

- sledování pokrytí kódu,
- interaktivní a posimulační prostředí pro debugování,
- nástroje pro porovnání průběhu více simulací,
- jednotlivé podčásti modelu mohou být zapsány v různých jazycích,
- možnost procházet obsah pamětí.

Simulátor umožňuje uživateli nastavit vstupy navrženého obvodu (tzv. testovací vektory) a sledovat generované výstupy. Uživatel může nastavit časový interval, po který má simulace běžet. Průběh simulace si uživatel může prohlédnout ve formě vlnových grafů. Samotné okno pro vizualizaci výsledků je interaktivní, lze jej přibližovat/oddalovat a tím měnit zobrazený časový interval.

2.3.2 GHDL

GHDL je volně šiřitelný simulátor VHDL. Vstupní popis přeloží a vytvoří spustitelný soubor, který provádí samotnou simulaci. GHDL však neprovádí syntézu návrhu. [3, 4]

Vstupní VHDL překládá přímo do strojové formy (tj. bez nutnosti překladu do mezikódu jako např. C nebo C++). Proto by výsledný kód měl být rychlejší než při překladu s mezikódem. Neprovádí ovšem příliš mnoho optimalizací. Sami tvůrci tvrdí, že výstupní programy GHDL nejsou příliš rychlé [2]. Simulátor umožňuje ve vstupních souborech volat funkce nebo procedury implementované v jiných jazycích než VHDL.

Pro překlad využívá překladač GCC. Vývojáři se snaží, aby verze GHDL korespondovala s aktuální verzí tohoto překladače. GHDL neobsahuje žádné vizualizační rozhraní. Dokáže však produkovat výstup např. ve formátu VCD, který lze zobrazit v jiných vizualizačních nástrojích (např. GTKWave).

VCD [1] je textový formát určený pro zaznamenávání změn hodnot jednotlivých prvků systému v průběhu simulace. Formát byl definován společně s jazykem Verilog a je podporován celou řadou simulátorů.

2.3.3 GTKWave

GTKWave [5] je vizualizační nástroj založený na knihovně GTK+. Zobrazuje průběhy simulací ve formě vlnových grafů (digitální i analogové) a podporuje širokou škálu vstupních formátů. Není určen pro interaktivní běh simulace, ale pro procházení výsledků simulace po jejím skončení.

GTKWave byl vyvinut pro zkoumání chování rozsáhlých systémů. Předpokládá se tedy práce s velkým množstvím signálů, proto obsahuje několik přístupů k filtrování relevantních informací. Lze filtrovat pouze sadu signálů, která uživatele aktuálně zajímá (např. pomocí regulárních výrazů). Navíc je způsob vizualizace výsledků rozšiřitelný. Příkladem může být zásuvný modul *TwinWave*, který umožňuje zobrazit průběhy dvou simulací současně.

Kapitola 3

Návrh simulátoru

Tato kapitola se věnuje návrhu jednotlivých komponent simulátoru a jejich interakci mezi sebou. Je zde popsána struktura interního simulačního modelu, simulační algoritmus, ale také způsob interakce s uživatelem a jednotlivé nástroje, které má k dispozici.

3.1 Specifikace požadavků

Jedním ze základních požadavků kladených na návrh simulátoru je, aby byl schopen pracovat se stejnou množinou vstupních modelů jako verifikační nástroj HADES. Není však nutné, aby simulátor implementoval sémantiku všech konstrukcí, které tento jazyk dovoluje (některé jsou z hlediska simulace nepodstatné). Nicméně jejich výskyt ve vstupním popisu modelu by neměl způsobit chybu. Navíc by část simulátoru zodpovědná za načítání vstupního modelu a sestavení interního modelu pro simulaci měla být rozšiřitelná o specifikace dalších jazyků. Simulátor by tedy neměl být vázán jen a pouze na modely VAM.

Při návrhu musí být kladen důraz především na:

- vytváření efektivní interní reprezentace vstupního modelu,
- efektivní provádění simulace.

Další požadavky jsou kladeny na vizualizaci výsledků:

- návrh simulátoru by měl umožnit přidávat nové způsoby vizualizace bez nutnosti zásahu do simulačního jádra,
- simulátor musí umožnit jak grafický, tak textový způsob procházení výsledků simulace,
- grafické rozhraní musí nabízet přehledné zobrazení průběhu simulace.

Skupina požadavků na řízení simulace:

- simulace musí být interaktivní – uživatel musí mít možnost simulaci řídit (např. pozastavení nebo posun v čase),
- uživateli musí být umožněno definovat počáteční stav modelu.

Poslední skupinou jsou požadavky na implementaci:

- implementace musí respektovat pravidla objektově orientovaného programování,
- simulátor musí být přenositelný mezi operačními systémy GNU/Linux a MS Windows.

3.2 Popis vstupního modelu

Vstupem simulátoru je graf toku dat popsaný v jazyce VAM [21], který je také vstupním jazykem pro verifikační nástroj HADES. Jelikož simulátor `dfsim` vzniká jako podpůrný nástroj pro HADES, byl tento jazyk zvolen jako referenční.

Jedná se o model a současně o jazyk schopný popisu modelů mikroprocesorů s jednou linkou zřetězení na úrovni přenosů dat mezi registry (*angl., registr transfer level*). Uzlem grafu může být úložiště (registr či paměť), nebo funkční obvod. Rozšiřuje tedy grafy toku dat o speciální typ uzlů, které jsou schopny uchovávat svou hodnotu. Jednotlivé uzly jsou mezi sebou propojeny pomocí orientovaných hran (signálů). Každý prvek grafu poskytuje sadu atributů, jejichž nastavením lze upravit jeho chování. Nejsou-li tyto atributy z hlediska simulace nutné, jsou ignorovány.

Syntaxe pro popis modelu VAM je podobná jazyku LISP. Model je popsán v jednom zdrojovém souboru, který obsahuje obecné informace o modelu, deklarace spojovacích signálů, úložišť a funkčních uzlů (v libovolném pořadí). Syntaxe deklarace a sémantika jednotlivých prvků a jejich atributů (pouze těch z hlediska simulace relevantních) je popsána dále v této sekci. Obecná struktura popisu modelu je:

```
(model jméno_modelu
       (sig ...) /* deklarace signálu */
       ...
       (reg ...) /* deklarace registru */
       ...
       (mem ...) /* deklarace paměťové jednotky */
       ...
       (fnode ...) /* deklarace funkčního uzlu */
       ...
)
```

Signály představují datové cesty mezi jednotlivými uzly s explicitně specifikovanou bitovou šířkou. Hodnotu signálu může zapisovat pouze jeden uzel, díky čemuž nedochází k vícenásobným zápisům v jednom simulačním kroku, což by mohlo vést k neočekávanému chování modelu. Na druhou stranu může libovolný počet uzlů hodnotu signálu číst, což často vede ke zjednodušení výsledného grafu.

Deklaraci signálu lze zapsat jako:

```
(sig identifikátor bitová-šířka)
```

Význam jednotlivých atributů je následující:

- `identifikátor` je unikátní identifikátor signálu,
- `bitová-šířka` je kladné celé číslo.

Úložiště (registr nebo paměť) je prvek grafu schopný uchovat svou hodnotu po předem nespecifikovanou dobu. Jednotlivé typy úložišť mají jasně vymezená rozhraní (názvy a sémantiku vstupních a výstupních signálů). Všechna úložiště mají nulové zpoždění při čtení hodnoty a jednotkové zpoždění při zápisu.

Deklarace registru má podobu:

```
(reg identifikátor bitová-šířka
     (d identifikátor_signálu)
     (q identifikátor_signálu)
     (we identifikátor_signálu)
     (stall identifikátor_signálu)
     (clr identifikátor_signálu)
)
```

Položky v tomto zápisu mají následující význam:

- `identifikátor` je unikátní identifikátor úložiště (registru nebo paměti),
- `bitová-šířka` je kladné celé číslo udávající bitovou šířku registru,
- `d, q, we stall, clr` definují signály připojené k daným portům registru,
- `d` je vstupní datový port registru,
- `q` je výstupní datový port registru,
- `we` je port registru pro povolení zápisu (není-li do něj zapojen žádný signál, je zápis vždy povolen),
- `stall` je port registru při jehož aktivaci je zápis do registru blokován,
- `clr` je port pro vynucené nulování registru,
- priorita signálů pro výpočet nového stavu je `stall > clr > we`.

Deklarace adresovatelného úložiště, tedy paměti, má podobu:

```
(mem identifikátor
     bitová-šířka-buňky
     bitová-šířka-adresy
     počet-paměťových-buněk
     (ports
          (port
             (dir {ro|wo})
             (timing zpoždění)
             (en signál)
             (addr signál)
             (data signál)
          )
     )
)
```

Význam jednotlivých položek je následující:

- `identifikátor` je unikátní identifikátor úložiště (registru či paměti),
- `bitová-šířka` je kladné celé číslo udávající bitovou šířku jedné paměťové buňky,
- `bitová-šířka-adresy` je kladné celé číslo udávající bitovou šířku adresy,
- `počet-paměťových-buněk` je kladné celé číslo udávající počet paměťových buněk,
- sekce `ports` definuje přístupové porty paměti (pro čtení či zápis),
- atribut `dir` určuje typ portu (`ro` pro čtení, `wo` pro zápis),
- atribut `timing` je nezáporné celé číslo určující zpoždění při přístupu (aktuálně je povolena 0 pro čtení a 1 pro zápis),
- `en` určuje povolovací signál,
- `addr` definuje signál pro adresaci daného portu,
- `data` je vstupní/výstupní datový signál (záleží, zda se jedná o port pro čtení/zápis).

Funkční obvod představuje funkci transformace vstupních dat na výstup. Každý výstupní signál může mít přiřazen jiný výraz pro vyhodnocení. Když je funkční uzel aktivován, jednotlivé výrazy jsou vyhodnoceny a provedou se patřičná přiřazení (tzn. každý výstupní signál může mít po aktivaci uzlu jinou hodnotu). Jak vyhodnocení výrazu, tak propagace výsledků na výstupní signály trvá nulový čas, proto modely VAM nesmí obsahovat cyklus složený pouze z funkčních uzlů.

Obecná deklarace funkčního uzlu má strukturu:

```
(fnode identifikátor
       (inputs seznam-vstupních-signálů)
       (outputs seznam-výstupních-signálů)
       (assign
           (:= výstupní-signál výraz)
           ...
       )
)
```

Význam atributů je následující:

- `identifikátor` je unikátní identifikátor funkčního uzlu,
- atribut `inputs` určuje seznam vstupních signálů (každý vstupní signál je argument funkce),
- atribut `outputs` určuje seznam výstupních signálů (každému výstupnímu signálu patří jedno přiřazení),
- sekce `assign` definuje jednotlivá přiřazení jako dvojice `(výstupní-signál, výraz)`,
- `výraz` definuje výraz zapsaný v prefixové notaci vyhodnocený při aktivaci uzlu.

```
(model counter
  (sig dataIn  4)
  (sig dataOut 4)
  (sig const_1 1)

  (fnode f_enable
    (input )
    (output const_1)
    (assign (:= const_1 1))
  )

  (fnode f_incr
    (input dataOut)
    (output dataIn)
    (assign
     (:= dataIn (+ dataOut 1))
    )

  (reg cnt 4
    (we const_1)
    (d dataIn)
    (q dataOut)
  )
)
```

Obrázek 3.1: Popis 4-bitového čítače v jazyku VAM.

Přesnější informace o modelu VAM, včetně popisu atributů, které nejsou při simulaci používány, lze nalézt v [21]. Ukázka zápisu modelu 4-bitového čítače je zobrazena na obrázku 3.1.

Popis modelu je načten a transformován do podoby abstraktního syntaktického stromu (*angl., abstract syntax tree – AST*). AST představuje popis struktury interního simulačního modelu (mezikrok mezi vstupním popisem a jeho interní reprezentací pro simulaci) a je nezávislý na zdrojovém jazyce. Ačkoli je jazyk VAM použit jako referenční, simulátor umožňuje přidat podporu pro nové vstupní jazyky. Nový překladač musí pouze dodržet obecnou strukturu AST. Sestavení interního simulačního modelu je pak řízeno algoritmem průchodu stromové struktury AST, který je implementován pomocí návrhového vzoru návštěvník (*angl., visitor pattern*) [14].

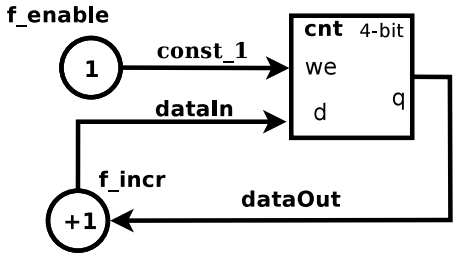
3.3 Návrh interního modelu

Struktura interního simulačního modelu je založena na struktuře modelu VAM. Přesto, jak již bylo uvedeno v předchozí sekci, není sestavení interního modelu na VAM přímo závislé. Interní model je, stejně jako model VAM, orientovaný graf, který se skládá z úložišť (registrů a pamětí), funkčních uzlů a signálů, které je propojují. Na nejvyšší úrovni jsou vstupní model VAM a jeho interní reprezentace pro simulaci izomorfní. Každý prvek modelu VAM má svou reprezentaci v simulačním modelu a vazby mezi nimi (signály) jsou také zachovány. Na nižší úrovni však dochází, pro potřeby simulace, k jistým modifikacím.

Všechny prvky grafu jsou klasifikovány do dvou skupin – *stavové* a *bezstavové*. Libovolný stavový prvek je schopen uchovat svou hodnotu (stav) po neomezenou dobu. Patří sem tedy registry a paměti. Také signály jsou schopny uchovávat svou hodnotu, a patří proto mezi stavové prvky. V rámci modelu je lze tedy chápat jako proměnné. Bezstavové prvky nejsou

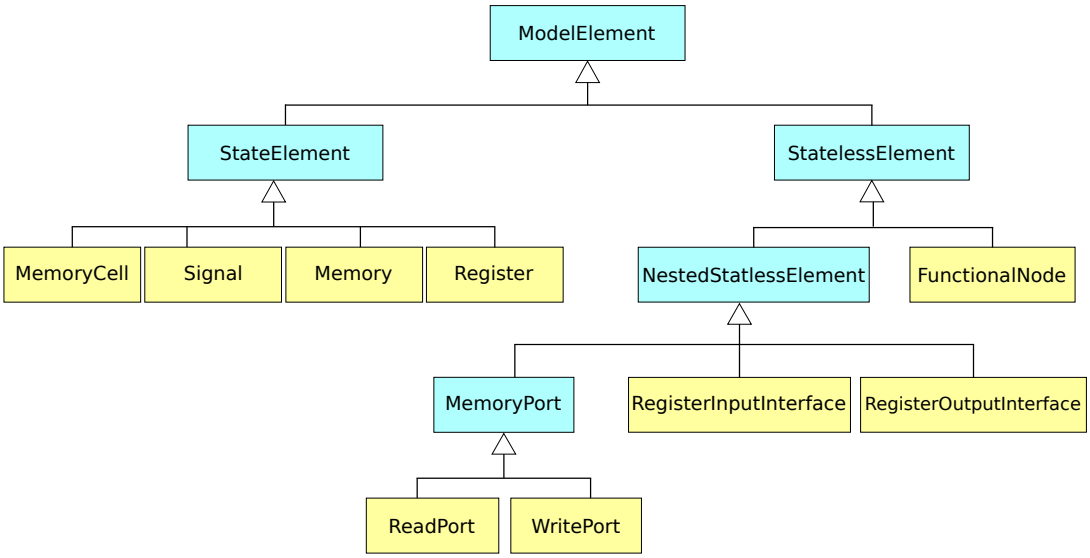
schopny udržet svůj stav, ale mohou být aktivovány a tím produkovat novou hodnotu na své výstupy.

Ukázka interní simulační reprezentace modelu 4-bitového čítače z obrázku 3.1 je zobrazena na obrázku 3.2. Kružnice reprezentují funkční uzly, obdélník je registr a orientované hrany, které je spojují, modelují signály.



Obrázek 3.2: Ukázka interní simulační reprezentace 4-bitového čítače.

Návrh interního modelu respektuje zásady objektově orientovaného programování. Jednotlivé třídy, které tvoří kompletní model, jsou podrobněji popsány v následujících sekcích. Hierarchie dědičnost tříd modelu je znázorněna na obrázku 3.3. Je zde vidět rozdělení prvků modelu na stavové a bezstavové. Modrou barvou jsou znázorněna pouze rozhraní nebo abstraktní třídy. Žluté jsou konkrétní třídy, jejichž instance tvoří model.



Obrázek 3.3: Diagram dědičnosti tříd modelu. Modrou barvou jsou znázorněna pouze rozhraní nebo abstraktní třídy.

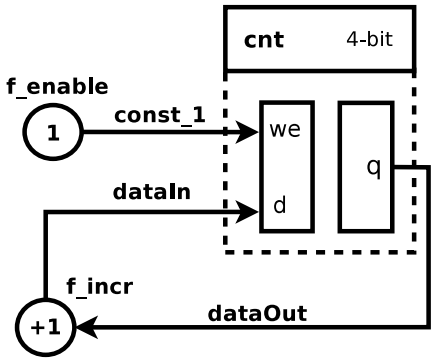
3.3.1 Dekompozice interního modelu

Na nejvyšší úrovni jsou vstupní VAM model a z něj sestavený interní simulační model izomorfní. Na nižší úrovni však dochází k určité dekompozici. Jelikož jsou úložiště stavové prvky a mají nenulové zápisové zpoždění, není jejich výstup přímo závislý na vstupu.

Každé úložiště v modelu je dekomponováno na samostatné bezstavové prvky, které představují jeho vstup/výstup. Jsou proto označovány jako vstupní/výstupní rozhraní. Jelikož

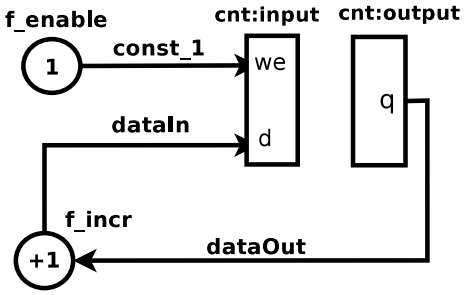
se jedná o bezstavové prvky, mohou být aktivovány a provést určitou operaci. Je-li aktivováno vstupní rozhraní, stav zdrojového úložiště může (ale nemusí) být modifikován. Vstupní rozhraní tedy představuje funkci příštího stavu úložiště. Při aktivaci výstupního rozhraní je stav zdrojového úložiště propagován na všechny výstupní signály.

Registr má typicky právě jedno vstupní a výstupní rozhraní. Vstupní rozhraní registru je množina všech jeho vstupních signálů. Obdobně výstupní rozhraní představuje sadu všech výstupních signálů. Vstupní/výstupní rozhraní u paměťových jednotek je reprezentováno pomocí portů pro zápis/čtení. Dekompozice registru je znázorněna na obrázku 3.4.



Obrázek 3.4: Ukázka dekompozice modelu 4-bitového čítače.

Původní úložiště jsou z grafu odebrána a dále slouží pouze k ukládání stavu. Pracovat s jejich stavem je však možné pouze prostřednictvím vstupního/výstupního rozhraní, kterými bylo dané úložiště nahrazeno. Výsledkem odebrání úložišť z grafu je nový graf, který se skládá pouze z bezstavových prvků a signálů, které je propojují. Dekomponovaný graf modelu 4-bitového čítače z obrázku 3.4 je demonstrován na obrázku 3.5.



Obrázek 3.5: Ukázka dekomponovaného modelu 4-bitového čítače obsahujícího pouze bezstavové prvky a signály.

Cílem dekompozice je získat graf, který lépe odráží skutečnost, že výstup úložišť není přímo závislý na jejich vstupu. Pro potřeby simulace je také výhodné, že se výsledný graf skládá pouze z bezstavových prvků, které mohou být aktivovány a produkovat tak nové hodnoty. Tím se interní model také více přiblíží grafům toku dat, které (ve své základní podobě) paměťové prvky neobsahují.

3.3.2 Evaluační plán modelu

Paradigma grafů toku dat umožňuje, aby byl libovolný uzel grafu aktivován kdykoli je na jeho vstupu dostatečné množství dat (více v sekci 2.2.1). Pro reálnou aplikaci je však

nutno zavést tzv. *evaluační plán (angl., schedule)*, který explicitně určuje pořadí vyhodnocení jednotlivých uzlů. Jelikož plán určuje pořadí *aktivace* uzlů, musí zahrnovat pouze bezstavové prvky grafu. Proto je plán sestaven až pro dekomponovaný graf. Pro sestavení plánu jsou klíčové následující vlastnosti:

1. vyhodnocení funkčního uzlu a propagace hodnoty přes signál je provedeno v nulovém čase,
2. zápis do úložišť má nenulové zpoždění,
3. hodnotu signálu může číst více uzlů, pouze jeden uzel však může hodnotu zapisovat,
4. uzel může mít více výstupních signálů,
5. výstupní hodnota uzlu je výhradně funkcí jeho vstupů – vedlejší efekty (např. globální proměnné) nejsou dovoleny, (jedinou výjimkou jsou rozhraní úložišť, které pracují se stavem daného úložiště),
6. uzel je vyhodnocen pouze pokud má na svých vstupech dostatečné množství dat,
7. dekomponovaný graf neobsahuje cyklus.

Vycházíme-li z vlastností 1 a 2, dojdeme k závěru, že v jednom časovém okamžiku dochází pouze ke změnám hodnot signálů (propagace hodnoty přes signál trvá nulovou dobu). Z vlastností 5 a 7 vyplývá, že počet těchto změn je konečný, protože stavy úložišť se během jednoho okamžiku změnit nemohou. Na začátku dalšího časového okamžiku (tj. po uplynutí zápisového zpoždění) dojde k aktualizaci stavů úložišť a proces propagace hodnot je opakován.

Aby bylo dosaženo efektivní simulace, je cílem evaluačního plánu zamezit opakovanému vyhodnocování uzlů během jednoho simulačního kroku. Jinak řečeno, každý uzel by měl být aktivován až tehdy, když všichni jeho předchůdci byli již aktivováni (tj. hodnoty všech vstupních signálů uzlu jsou platné pro daný simulační krok). Proto je na graf aplikován algoritmus topologického řazení [19], jehož výstupem je posloupnost uzlů, v níž je každý uzel grafu zařazen až za všechny své předchůdce. Tato posloupnost představuje evaluační plán simulačního modelu.

Algoritmus 1 demonstruje zjednodušenou verzi algoritmu topologického řazení používanou pro sestavení evaluačního plán. Nejdříve jsou z grafu vybrány ty uzly, které nemají žádné vstupní hrany (jejich výstup není závislý na jiném uzlu) a jsou vloženy do výsledné sekvence (vstupní graf musí obsahovat alespoň jeden takový prvek). Následně jsou tyto uzly a všechny jejich výstupní signály z grafu odstraněny. Tím vzniknou nové uzly bez vstupních signálů. Tento postup se opakuje, dokud vstupní graf obsahuje alespoň jeden uzel, který nemá žádný vstup. Pokud již není možné postup opakovat a graf stále obsahuje alespoň jednu hranu, byl detekován cyklus a graf není topologicky seřaditelný.

Velkou výhodou takto sestaveného evaluačního plánu je to, že je možné jej sestavit před samotnou simulací (pouze jednou). Nezvyšuje se tedy režie během simulace. Navíc jeho úspěšným sestavením je zajištěno, že vstupní model je sémanticky v pořádku (např. neobsahuje cyklus), což je úloha, kterou by simulátor stejně musel provádět.

Hlavní výhodou je ovšem fakt, že zamezuje redundantnímu vyhodnocování uzlů. Budou-li uzly grafu vyhodnocovány v pořadí daném evaluačním plánem, pak každý uzel bude aktivován až tehdy, kdy jsou již vypočteny hodnoty všech jeho vstupů. Není tedy nutné

```
Input: Interní model
Output: Evaluační plán
U ← množina všech uzlů grafu;
S ← seznam obsahující uzly určené ke zpracování;
L ← seznam obsahující výslednou sekvenci;
for uzel u z množiny U do
    if uzel u nemá žádný vstupní signál then
        Přidej u do S;
while seznam S je neprázdný do
    Odeber první prvek z S a ulož jej do n;
    Přidej n na konec L;
    for každý uzel m, který je signálem přímo spojený s uzlem n do
        Odeber spojovací signál z grafu;
        if uzel m již nemá další vstupní signál then
            Vlož uzel m na konec S;
if graf obsahuje alespoň jednu hranu then
    Vrať chybu – graf obsahuje cyklus;
else
    Vrať L jako výslednou sekvenci;
```

Algoritmus 1: Zjednodušená verze algoritmu topologického řazení používaného při sestavení evaluačního plánu.

procházet grafem, aby byla hodnota vypočítána. To lze demonstrovat na příkladu z obrázku 3.5. Vstupní rozhraní registru `cnt:input` je aktivováno tehdy a jen tehdy, když byly uzly `f_incr` a `f_enable` již vyhodnoceny. Stejně tak uzel `f_incr` je aktivován až poté, co jeho jediný předchůdce – výstupní rozhraní registru – `reg:output` byl aktivován. Ten společně s uzlem `f_enable` nemá žádné vstupy, proto může být vyhodnocen jako první. Při vyhodnocování uzlu není tedy nutné procházet grafem, aby se získala správná hodnota vstupu, protože tato hodnota již byla plně spočítána dříve.

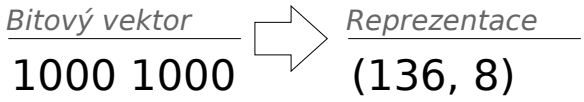
Jak bylo popsáno v sekci 2.2.1, grafy toku dat jsou řízeny přítomností dat na vstupních hranách uzlů, nikoliv časem. Nicméně pro potřeby simulace je nutno zavést také časovou doménu. Pro její zavedení je klíčové především nenulové zápisové zpoždění úložišť. Vzhledem k vlastnostem 5 a 7 dojde během jednoho simulačního kroku ke konečnému počtu změn hodnot signálů v grafu. Musí tedy nastat situace, kdy již není možné změnit další hodnotu. Evaluační plán navíc usměrňuje průběh vyhodnocování uzlů tak, že změny hodnot signálů v grafu ustanou poté, co je vyhodnocen poslední uzel z plánu. Další změny hodnot jsou možné až po uplynutí zápisového zpoždění úložišť, kdy dojde k aktualizaci jejich stavu. Proto je za *jednotku času* během simulace označována jedna iterace evaluačním plánem.

3.3.3 Reprezentace hodnot v modelu

Každý stavový prvek v modelu musí mít explicitně specifikovanou svou bitovou šířku, která udává největší možnou hodnotu, jakou je schopen uchovat. Všechny hodnoty jsou výhradně celá čísla bez znaménka. Návrh simulátoru ovšem počítá s budoucím rozšířením o čísla se znaménkem a přidáním nových operací ve znaménkové aritmetice.

Hodnoty jsou definovány na určitém počtu bitů, jeví se tedy jako bitové vektory. Na nižší úrovni jsou ovšem reprezentovány jako dvojice *(hodnota, bitová_šířka)*. Dvojice je reprezen-

tována třídou `BitLimitedInteger`, která navíc definuje množinu operací nad tímto typem. Jedná se o *neměnitelný datový typ*. Po vytvoření instance již není možné měnit její atributy. Navíc výsledkem každé operace nad tímto typem je nová instance – nedochází tedy k modifikaci původní instance, což je důležité především při uchovávání historie stavů jednotlivých stavových prvků (více v sekci 3.5.2).



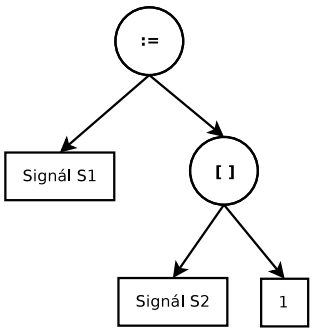
Obrázek 3.6: Reprezentace bitového vektoru pomocí uspořádané dvojice.

3.3.4 Vyhodnocování výrazů

Výrazy jsou v modelu reprezentovány jako výrazové stromy. Jsou sestaveny staticky během transformace AST vstupního modelu do podoby interního simulačního modelu (viz sekce 3.2). Vyhodnocovány jsou za běhu simulace když dojde k aktivaci funkčního uzlu. Výslednou hodnotu vyhodnocení stromu ovlivňují listové uzly, které mohou reprezentovat buď konstantu, nebo signál, jehož hodnota odpovídá aktuálnímu stavu daného signálu.

Výsledkem vyhodnocení výrazového stromu je bitový vektor. Bitové šířky mezivýsledků se mohou lišit od šířky původních operandů. Během vyhodnocování totiž nelze předem určit, na kolika bitech má být reprezentován finální výsledek. Aby nedocházelo ke ztrátě informace (např. ořezáním mezivýsledku na určitý počet bitů), je bitová šířka výsledků operací nastavena vždy tak, aby byla dostačující pro danou hodnotu. Výsledky jsou ořezány/rozšířeny až při přiřazení na konkrétní stavový prvek.

Jednoduchá ukázka výrazového stromu je zobrazena na obrázku 3.7. Strom při vyhodnocení nastaví hodnotu signálu `S1` na hodnotu prvního bitu (číslováno od nuly) signálu `S2`.



Obrázek 3.7: Ukázka výrazového stromu, jehož vyhodnocením dojde k nastavení hodnoty signálu `S1` na hodnotu prvního bitu (číslováno od nuly) signálu `S2`.

Podporované operace

Aktuálně je podporována podmnožina operací modelu VAM (výčet všech operací lze nalézt v [21]). Oproti VAM nejsou podporovány operace ve znaménkové aritmetice. Podporované operace lze rozdělit do následujících skupin: (i) aritmetické, (ii) bitové, (iii) logické, (iv) vektorové, (v) relační, (vi) podmíněné výrazy.

3.4 Simulační jádro

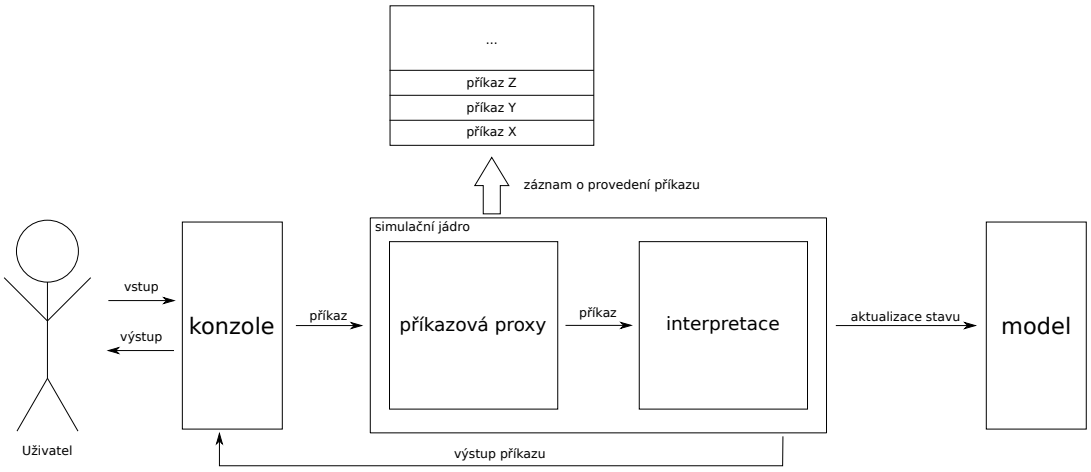
Simulační jádro má na starosti řízení procesu simulace, zpracování a uchování informací o běhu a jejich poskytnutí uživateli k prezentaci. Navíc slouží také jako interpret uživatelských příkazů. Existují dvě skupiny příkazů – pro řízení běhu simulace a pro práci se stavem modelu (získávání hodnoty jednotlivých prvků, případně její modifikace). Výčet podporovaných příkazů a jejich popis je v sekci 3.4.1.

Důležitou vlastností simulačního jádra je nezávislost na způsobu prezentace dat. Jedná se o simulační framework, k němuž mohou být připojena různá výstupní rozhraní. Je v režii každého výstupního rozhraní, jakým způsobem data prezentuje uživateli. Možnosti prezentace dat jsou podrobněji popsány v sekci 3.6.

3.4.1 Podporovaná sada příkazů

Simulační jádro je schopno interpretovat sadu příkazů, kterými uživatel může řídit běh simulace nebo pracovat se stavem modelu. Zadávání příkazů a jejich provádění je navrženo podle návrhového vzoru *příkaz* [14]. Každý typ příkazu je reprezentován samostatnou třídou, která zapouzdřuje jeho argumenty. Tímto způsobem je možno evidovat, které příkazy byly v jakém pořadí zadány a následně je exportovat v serializované podobě pro pozdější spuštění.

Příkazy jsou zadávány skrze tzv. *příkazovou proxy*. Jedná se o součást simulačního jádra, která zaznamenává jednotlivé příkazy a interpretuje je. Schéma interakce uživatele a simulačního jádra při zadávání příkazů je znázorněno na obrázku 3.8.



Obrázek 3.8: Průběh zpracování uživatelského příkazu.

Aktuálně jsou podporovány následující příkazy:

- `initialize [soubor_s_počátečními_stavy_úložišť]` – Provede inicializaci simulačního modelu. Není-li vstupní soubor zadán, všechna úložiště jsou vynulována.
- `reinitialize [t=0]` – Provede re-inicializaci simulačního modelu do stavu, který měl v čase `t`.
- `run N` – Simulační jádro provede `N` simulačních kroků.

- `step` – Simulační jádro provede jeden simulační krok.
- `stop` – Běží-li simulace, je zastavena (jinak nemá žádný efekt). Příkaz je vhodný v případě, že simulace běží v samostatném procesu/vlákně.
- `dumpsig jméno [t=aktuální_simulační_čas] [format`[1]`]` – Vrací hodnotu signálu dle zadaného jména, kterou nabýval v čase `t`.
- `dumpreg jméno [t=aktuální_simulační_čas] [format`[1]`]` – Vrací hodnotu registru dle zadaného jména, kterou nabýval v čase `t`.
- `dumpmem jméno [t=aktuální_simulační_čas] [format`[1]`]` – Vrací hodnoty všech buněk paměti dle zadaného jména, kterou nabývaly v čase `t`.
- `dumpcell jméno_paměti adresa_buňky [t=aktuální_simulační_čas] [format`[1]`]` – Vrací hodnotu paměťové buňky dle zadaného jména a adresy, kterou nabývala v čase `t`.
- `setreg jméno hodnota`[2] – Nastaví hodnotu registru (dle jména) na zadanou hodnotu a přepočítá poslední simulační krok s novými hodnotami.
- `setmem jméno hodnota`[2] – Nastaví hodnotu všech buněk paměti (dle jména) na zadanou hodnotu a přepočítá poslední simulační krok s novými hodnotami.
- `setcell jméno_paměti adresa_buňky hodnota`[2] – Nastaví hodnotu buňky (dle adresy) paměti (dle jména) na zadanou hodnotu a přepočítá poslední simulační krok.
- `output arg1 ...argN` – Opíše na výstup jednotlivé argumenty. Příkaz je užitečný především při vytváření automatizovaných skriptů.
- `export jméno_souboru` – Exportuje dosavadní průběh simulace do zadaného souboru ve formátu VCD.

3.5 Proces simulace

Proces simulace se skládá ze dvou fází – inicializace a běhu. V obou fázích se využívá řada datových struktur. Některé jsou potřebné pro běh či řízení simulace, jiné slouží převážně pro optimalizaci běhu. Mimo samotný model a jeho evaluační plán, pracuje simulační jádro s *kalendářem událostí* a *záznamníkem změn stavu modelu*. Simulační jádro je navrženo tak, že žádná z používaných datových struktur není globální. Teoreticky je tak možné rozšířit simulátor o podporu souběžných simulací a porovnání jejich běhu.

3.5.1 Kalendář událostí

Kalendář událostí je datová struktura používaná v diskrétních simulacích. Slouží k plánování událostí na konkrétní simulační čas a k jejich provedení, jakmile je tento čas dosažen.

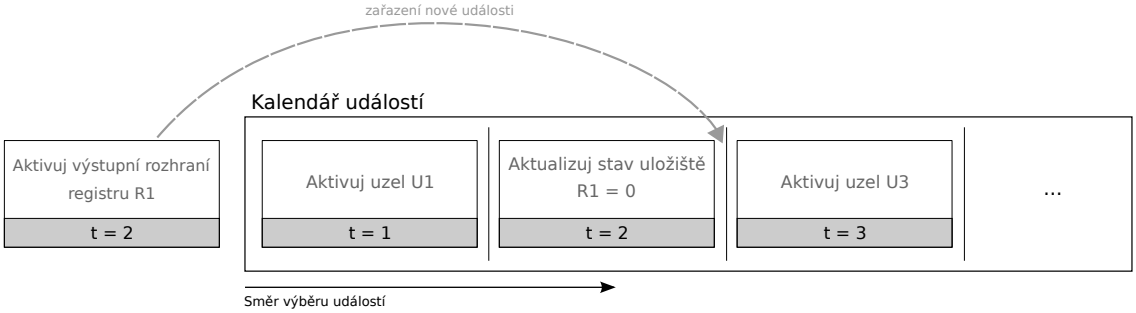
[1] parametr `format` určuje formát výstupu hodnoty. Může nabývat hodnot `bin, oct, dec, hex` pro výstup v binární, osmičkové, desítkové a hexadecimální soustavě. Není-li tento argument zadán, výstupní hodnota je ve tvaru `(_ hodnota bitová_šířka)`.

[2] Hodnoty je možno zadávat v binární, osmičkové a šestnáctkové soustavě. K zadané hodnotě je nutno přidat prefix `0b`, `0o` nebo `0x`. Bez prefixu je zapsána hodnota v desítkové soustavě.

Simulační jádro definuje minimální rozhraní, které musí každý kalendář událostí implementovat, aby jej bylo možné použít během simulace. Samotná implementace se pak může lišit.

V diskrétních simulacích je typické použití *prioritního kalendáře událostí*, který je schopen řadit události nejen podle času, ale také podle jejich priority. V případě prezentovaného simulátoru nejsou priority událostí podstatné, proto jsou události řazeny pouze podle zadaného času a podle pořadí, v jakém jsou plánovány.

Jelikož při simulaci nedochází k plánování výrazného množství událostí, používá simulační jádro kalendář implementovaný pomocí datové struktury seznam. Jedná se o jednoduchou variantu, která však v tomto případě dostačuje (nejsou kladeny výrazné požadavky na rychlost plánování událostí). Princip kalendáře událostí je demonstrován na obrázku 3.9.



Obrázek 3.9: Princip použití kalendáře událostí.

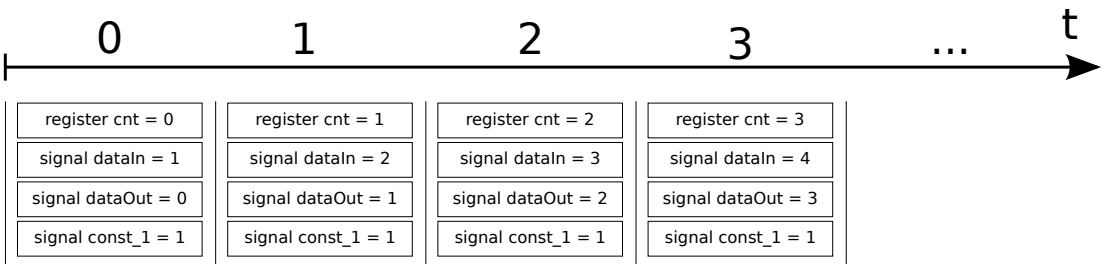
Momentálně jsou podporovány dva typy událostí – pro změnu stavu úložiště a pro vynucenou aktivaci uzlu grafu. Událost aktualizace stavu úložiště je potřebná kvůli nenulovému zpoždění při zápisu. Rozhodne-li funkce příštího stavu úložiště v čase t, že má dojít ke změně stavu daného úložiště, pak je na čas $t+<zpoždění\_při\_zápisu>$ naplánována tato událost. Událost vynucené aktivace uzlu je pak spojena především s optimalizací evaluačního plánu, kdy jsou uzly bez vstupních hran z plánu vyřazeny a je potřeba jejich aktivaci explicitně naplánovat.

3.5.2 Záznamník historie stavu modelu

Tato datová struktura slouží k uchování historie stavů jednotlivých stavových prvků během simulace. Je tedy možné zpětně zjistit, jaký byl stav konkrétního prvku v libovolném čase. Uchovávají se hodnoty všech signálů, registrů a paměťových buněk.

Jelikož je simulační jádro nezávislé na způsobu prezentace dat, jsou všechny hodnoty ukládány ve formě bitového vektoru (popsáno v sekci 3.3.3), což je přirozený způsob reprezentace dat v modelu. Protože se jedná o neměnitelný datový typ a výsledkem operací nad ním je vždy nová instance, je možné hodnoty bezpečně ukládat (nemůže dojít k modifikaci vnitřních atributů konkrétní instance). Princip záznamníku je demonstrován na obrázku 3.10.

Momentálně je záznamník navržen jako seznam slovníků, kde index seznamu odpovídá simulačnímu času a slovníky obsahují hodnoty, které jsou zaznamenávány. Je zde prostor pro optimalizaci paměťových nároků. Ve fázi návrhu se momentálně nachází alternativní verze záznamníku, který zaznamenává pouze ty prvky modelu, jejichž hodnota se v daném čase změnila.



Obrázek 3.10: Demonstrace principu záznamníku stavu modelu.

3.5.3 Inicializace simulace

Během fáze inicializace dochází k nastavení počátečních hodnot jednotlivých úložišť, vynulování modelového času, inicializaci (vyprázdnění) datových struktur jako je kalendář událostí a záznamník změn stavu modelu.

Implicitní počátečního hodnota všech úložišť (ať už se jedná o registr či paměťovou buňku) je nula. Implicitní hodnota však může být explicitně změněna aplikováním speciálního vstupního souboru, který obsahuje nové počáteční hodnoty. Každý řádek tohoto souboru má tvar trojice `typ jméno hodnota` a určuje novou počáteční hodnotu konkrétního úložiště[3]. Na soubor je možno pohlížet jako na sekvenční program, kde řádky jsou příkazy, které jsou postupně interpretovány.

Uvažujme příklad modelu, který obsahuje dva registry (`registr1` a `registr2`) a jednu paměťovou jednotku `memory` se čtyřmi paměťovými buňkami. Aplikujeme-li soubor s počátečními hodnotami jako na obrázku 3.11, pak `registr2` bude mít implicitní počáteční hodnotu `0` a `registr1` hodnotu `1`. Paměťové buňky na adresách `0` a `1` budou mít hodnoty `0` a `1` (v uvedeném pořadí), zatímco buňky na adresách `2` a `3` budou obsahovat hodnotu `3`.

```
reg  register1 1
mem  memory    3
cell memory 0  0
cell memory 1  1
```

Obrázek 3.11: Ukázka souboru s počátečními hodnotami.

Po nastavení počátečních hodnot dochází k optimalizaci evaluačního plánu. Jsou z něj odstraněny všechny uzly, které nemají žádný vstup (každý graf musí mít alespoň jeden takový uzel, jinak není topologicky seřaditelný). Namísto opakované aktivace odebraného uzlu v každé iteraci průchodu evaluačním plánem je jejich aktivace explicitně naplánována do kalendáře událostí pouze na první simulační okamžik (tj. čas 0).

Odstraněný uzel může být buď funkční uzel, nebo výstupní rozhraní úložiště. Nemá-li funkční uzel vstupy, pak nikdy nemůže změnit svou výstupní hodnotu a je zbytečné jej opakovaně vyhodnocovat. Výstupní rozhraní registru svou výstupní hodnotu sice změnit může, ale vždy společně se změnou hodnoty odpovídajícího úložiště. Nastane-li tato situace, pak je aktivace výstupního rozhraní explicitně naplánována do kalendáře událostí.

[3]Při nastavování hodnoty konkrétní paměťové buňky se jedná o čtveřici `typ jméno_paměti adresa_buňky hodnota`

3.5.4 Běh simulace

Jakmile je dokončena fáze inicializace, samotná simulace může začít. Jedná se o kombinovanou simulaci, v níž jsou sloučeny principy diskrétní a spojité simulace. Algoritmus je navržen specificky pro simulační model s cílem provádět simulaci efektivně. Silně stojí na použití evaluačního plánu (sekce 3.3.2), který značně eliminuje redundantní vyhodnocování uzlů. Samotný algoritmus pak za běhu provádí další eliminaci zbytečných vyhodnocení.

Každý typ uzlu má jinou *aktivační podmínku* a *aktivační rutinu*. Aktivační rutina je operace, která je provedena při aktivaci uzlu. Při aktivaci funkčního uzlu dojde k vyhodnocení definovaných výrazů a propagaci výsledků na výstup. Je-li aktivováno vstupní rozhraní úložiště, může dojít k aktualizaci stavu odpovídajícího úložiště. Naopak aktivací výstupního rozhraní dojde k propagaci stavu úložiště na výstup. Každý uzel je aktivován tehdy a jen tehdy, je-li platná jeho aktivační podmínka. Například funkční uzel je aktivován pouze pokud se změnila hodnota alespoň jednoho jeho vstupu (od předchozího simulačního kroku).

```
t ← aktuální simulační čas;
while t < KONCOVÝ_ČAS do
    for každou událost u naplánovanou v kalendáři na čas t do
        Odeber událost u z kalendáře;
        Proveď událost u;
    for Každý uzel n z evaluačního plánu do
        if n je funkční uzel then
            if hodnota libovolného vstupu se změnila then
                Vyhodnoť funkci uzlu n;
                if hodnota výstupu se změnila then
                    Propaguj novou hodnotu na výstupní signály;
        else if n je výstupní rozhraní úložiště then
            if stav úložiště se změnil then
                Propaguj novou hodnotu na výstupní signály;
        else if n je vstupní rozhraní úložiště then
            if hodnota libovolného vstupu se změnila then
                Vyhodnoť funkci dalšího stavu úložiště;
                if hodnota úložiště by se měla změnit then
                    Naplánuj událost změny stavu daného úložiště do kalendáře na čas
                    t+zpoždění_zápisu;
                    Naplánuj událost aktivace výstupního rozhraní daného úložiště
                    do kalendáře na čas t+zpoždění_zápisu;
        Zaznamenej hodnotu uzlu n v čase t do záznamníku změn stavu modelu;
    t ← t + 1;
```

Algoritmus 2: Zjednodušená verze simulačního algoritmu.

Algoritmus 2 demonstruje zjednodušenou verzi simulačního algoritmu. V každém simulačním kroku jsou nejdřív provedeny všechny události, které jsou v kalendáři naplánovány na aktuální simulační čas. Následně proběhne iterace přes všechny uzly (optimalizovaného) evaluačního plánu. Před samotnou aktivací každého uzlu je zkontrolována platnost jeho aktivační podmínky. Je-li platná, dojde k aktivaci uzlu a tedy provedení odpovídající aktivační rutiny[4].

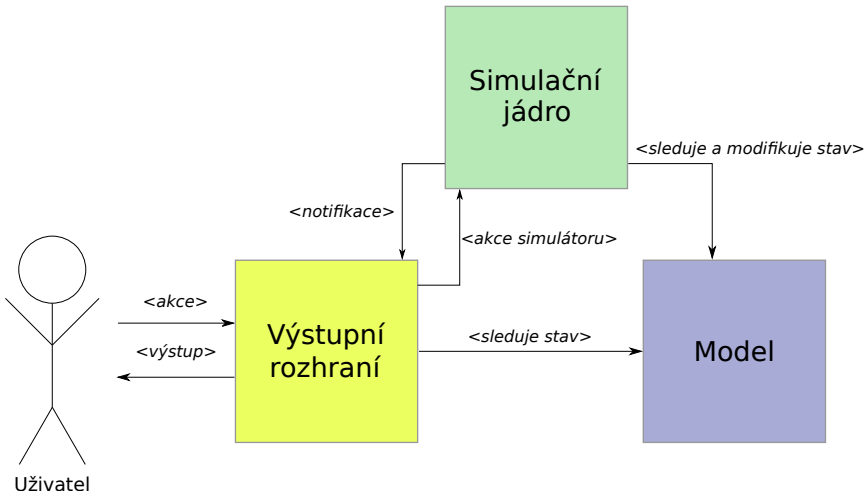
[4] Aktivační rutiny vstupních rozhraní registru a paměťových jednotek se mírně liší.

Navíc jsou před a po každém simulačním kroku vyvolány speciální funkce, které slouží k notifikaci výstupního rozhraní o stavu průběhu simulace. Mohou se lišit pro každé výstupní rozhraní – v textovém režimu řídí výstup simulace do souboru, zatímco v grafickém módu určují, kdy se má výstupní okno překreslit.

3.6 Prezentace výsledků

Průběh simulace je nezávislý na způsobu prezentace dat. Jedná se o simulační framework, ve kterém je možné připojit k simulačnímu jádru libovolné výstupní rozhraní. Je v režii každého výstupního rozhraní, jakým způsobem poskytne data uživateli. Jelikož vznikají oddělená výstupní rozhraní, může se každé plně soustředit na pokrytí konkrétních požadavků. Nemusí se omezovat nároky/požadavky ostatních výstupních rozhraní. Cílem je poskytnout uživateli různé pohledy na průběh simulace. Jedinou podmínkou je, aby výstupní rozhraní implementovalo minimální potřebný protokol pro komunikaci se simulačním jádrem.

Struktura simulátoru je inspirována návrhovým vzorem *Model–View–Presenter* [10]. Částečně se však liší v rolích, které jednotlivé části simulátoru zastávají. Struktura je podrobně znázorněna na obrázku 3.12. Model pouze uchovává svůj aktuální stav (jeho historii zaznamenává simulační jádro) a reprezentuje modelovaný systém. Roli presenteru zde zastává simulační jádro, které přímo manipuluje se stavem modelu a notifikuje výstupní rozhraní. K notifikaci typicky dochází před a po každém simulačním kroku. Je pak v režii výstupního rozhraní, jak na notifikaci zareaguje. V roli `View` stojí výstupní rozhraní. Jako jediný prvek v celé architektuře přijímá uživatelské akce, volá odpovídající akce simulátoru a prezentuje data uživateli.



Obrázek 3.12: Architekturální návrh simulátoru.

Dosud byla implementována tři samostatná výstupní rozhraní. První rozhraní poskytuje přímý výstup simulace v textové podobě. K němu existuje také interaktivní varianta, která umožňuje zadávat příkazy a spouštět uživatelské skripty. V poslední řadě simulátor poskytuje plnohodnotné grafické rozhraní pro vizualizaci průběhu simulace.

3.6.1 Textové výstupní rozhraní

Toto rozhraní produkuje přímý výstup simulace v textové podobě ve formátu VCD. Poskytuje uživateli jednoduché ovládání simulátoru v příkazové řádce, ale nenabízí interaktivní nástroje (příkazy). Uživatel při spuštění určí vstupní model (případně jeho počáteční stav) a po jakou dobu má simulace běžet. Po spuštění je proveden zadaný počet simulačních kroků a následně je simulátor ukončen. Proto je toto rozhraní vhodné především v situacích, kdy uživatele zajímá pouze přímý výstup simulace a nemá zájem do běhu zasahovat. Průběh simulace 4-bitového čítače po třech simulačních krocích je znázorněn na obrázku 3.13.

```
$date 02-05-2015 23:45:22 $end
$version dfsim v0.0.1 $end
$scope module signals $end
$var wire 1 const_1 const_1 $end
$var wire 4 dataIn dataIn $end
$var wire 4 dataOut dataOut $end
$upscope  $end
$scope module registers $end
$var reg 4 cnt cnt $end
$upscope  $end
$scope module memories $end
$upscope  $end
$enddefinitions  $end

#0
b0000 cnt
b1 const_1
b0001 dataIn
b0000 dataOut
#1
b0001 cnt
b0010 dataIn
b0001 dataOut
#2
b0010 cnt
b0011 dataIn
b0010 dataOut
```

Obrázek 3.13: Výstup simulace 4-bitového čítače po třech simulačních krocích.

Dalším cílem tohoto výstupního rozhraní je umožnit uživateli použít výsledky simulace v jiných analyzačních či simulačních nástrojích. To je díky použití výstupního formátu VCD, který je široce podporován, možné.

3.6.2 Interaktivní textové výstupní rozhraní

K textovému výstupnímu rozhraní z předchozí sekce existuje také interaktivní varianta, která umožňuje uživateli zadávat příkazy a řídit tak běh simulace nebo pracovat se stavem modelu. Toto rozhraní je také určeno pro použití v příkazové řádce. Má formu konzole, skrze kterou uživatel může zadávat příkazy a sledovat jejich výstupy. Výčet podporovaných uživatelských příkazů je uveden v sekci 3.4.1.

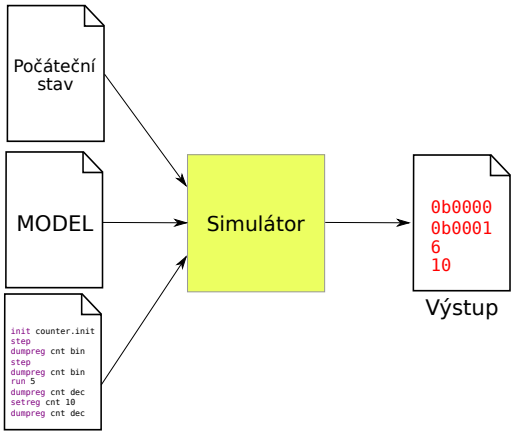
Interaktivita tohoto výstupního rozhraní je demonstrována na obrázku 3.14. Nejdříve je příkazem `init` načten počáteční stav modelu z externího souboru. Příkazem `step` dojde k provedení jednoho simulačního kroku. Následně je získán stav registru `cnt` v binární podobě. Po zadání příkazu `run 5` je provedeno pět simulačních kroků. Na konci příkladu je vidět explicitní vnucení stavu registru `cnt` pomocí příkazu `setreg`.

```
>>> init counter.init /*Inicializace modelu ze souboru*/
>>> step /*Provede jeden simulační krok*/
>>> dumpreg cnt bin /*Získá binární hodnotu registru cnt*/
0b0000
>>> step
>>> dumpreg cnt bin
0b0001
>>> run 5 /*Provede pět simulačních kroků*/
>>> dumpreg cnt dec
6
>>> setreg cnt 10 /*Nastaví hodnotu registru cnt na 10*/
>>> dumpreg cnt dec
10
```

Obrázek 3.14: Ukázka použití interaktivního textového výstupního rozhraní.

Hlavní rozdíl oproti přímému textovému výstupu je to, že rozhraní běží ve smyčce a musí být explicitně ukončeno (neinteraktivní varianta ukončí simulaci po provedení zadaného počtu kroků). Aby nedocházelo k zahrnutí uživatele spoustou informací, není výstup simulace implicitně produkován na výstup. Uživatel se tak může lépe orientovat ve výstupu zadaných příkazů. Nicméně výstup simulace (opět ve formátu VCD) je možné získat kdykoli za běhu explicitním zadáním příkazu `export`.

Další důležitou úlohou tohoto výstupního rozhraní je umožnit uživateli vytvářet a spouštět skripty. Uživatel může zapsat příkazy do souboru jako sekvenční program, který při spuštění předá simulátoru na vstup. Ten postupně jednotlivé příkazy interpretuje. Uživatel pak může analyzovat produkovaný výstup. Rozhraní je tedy vhodné pro vytváření automatizovaných testů vstupních modelů. Na základě výstupu spuštěného skriptu lze např. zkoumat, zda se dva modely chovají (pro daný skript) stejně. Jelikož skripty je možné spouštět opakovaně a není nutno je provádět ručně, dochází k výraznému snížení časových nároků.



Obrázek 3.15: Princip spouštění skriptů v interaktivním režimu.

Obrázek 3.15 ukazuje typický postup pro spuštění skriptu. Na vstup simulátoru zadá uživatel vstupní model (případně jeho počáteční stav) a skript, který se má provést. Výstupem je sekvence výstupů jednotlivých příkazů. Další analýza výstupu je již v režii uživatele, což poskytuje značnou volnost ve způsobu zpracování.

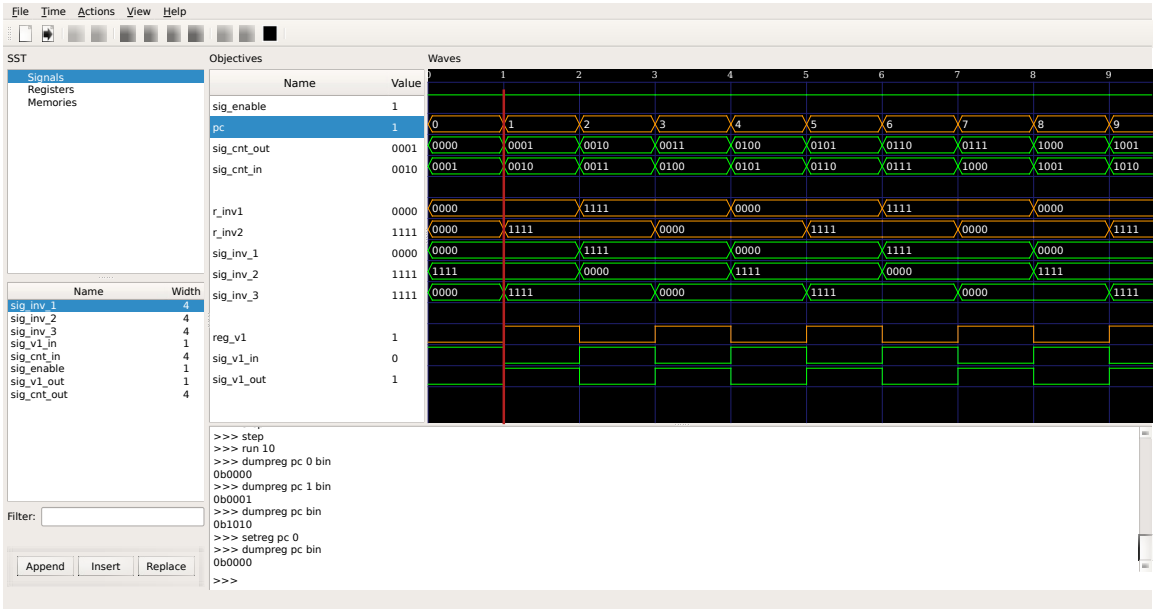
3.6.3 Grafické výstupní rozhraní

V tomto režimu výstupu vytváří simulátor plnohodnotné grafické rozhraní s cílem vizualizovat průběh simulace. Změny stavu jednotlivých prvků v čase jsou prezentovány formou digitálních vlnových grafů, což je na poli simulace hardware zažitý koncept. Uživatelské rozhraní je inspirováno vizualizačním nástrojem GTKWave.

Tím, že má uživatel možnost vidět průběh simulace, je schopen snadněji pochopit chování simulovaného modelu – což je také primární účel tohoto výstupního režimu. Mimo jiné má uživatel k dispozici nástroj pro manuální hledání chyb v chování modelu.

Rozhraní poskytuje uživateli interaktivní konzoli pro zadávání příkazů. V tomto případě však není cílem poskytnout uživateli nástroj pro spouštění skriptů, ale nabídnout co nejširší škálu nástrojů pro analýzu stavu modelu a řízení běhu simulace. Mimo to je k dispozici celá řada dalších nástrojů, které mají uživateli usnadnit analýzu modelu. Příkazy, které je možno zadat do konzole, je možné spustit také pomocí grafických prvků (buď položkou v některém z menu nebo pomocí ikony v panelu nástrojů).

Uživatel má možnost měnit formát (ASCII, binární, osmičkový, decimální a hexadecimální) zobrazených hodnot a pro větší přehlednost měnit barvu jednotlivých průběhů. Stejně jako interaktivní textové rozhraní umožňuje exportovat průběh simulace ve formátu VCD. Navíc dovoluje exportovat průběh v grafické podobě – buď jako bitmapový obrázek nebo vektorově ve formátu PDF. Kompletní uživatelské rozhraní včetně vstupní konzole, panelů pro výběr položek k vizualizaci a panelu nástrojů je ukázáno na obrázku 3.16.



Obrázek 3.16: Zobrazení kompletního uživatelského rozhraní.

Kapitola 4

Implementace vybraných částí simulátoru

Implementačním jazykem simulátoru je `Python3` (vyvíjeno na verzi 3.4, testováno také pod verzí 3.3). Byl zvolen především proto, že umožňuje rychlý vývoj a prototypování, ale současně je vhodný také pro vývoj i rozsáhlých aplikací. Jeho standardní knihovna je velmi rozsáhlá, díky čemuž je možné vyvarovat se rozsáhlým závislostem na knihovnách třetích stran. Momentálně simulátor využívá pouze knihovny `PLY` [9] pro vytvoření překladače vstupního jazyka VAM a `PyQt5` [8] pro vytvoření grafického rozhraní pro vizualizaci simulace. Interprety tohoto jazyka jsou dostupné na všech větší platformách, což uspokojuje důležitý požadavek na implementaci – přenositelnost mezi operačními systémy GNU/Linux a MS Windows.

4.1 Překladač vstupního modelu VAM

Překladač vstupního jazyka VAM, který transformuje popis vstupního modelu na abstraktní syntaktický strom, z něhož je pak sestaven simulační model, je implementován pomocí modulu `PLY` [9]. Jedná se o implementaci nástrojů `Lex` [17] a `Yacc` [15] čistě v jazyce python. Jde tedy o generátor překladačů jazyků, pro které uživatel definuje lexikální jednotky a gramatiku. Nicméně simulátor není na jazyku VAM, ani na překladačích vytvářených pomocí `PLY` nijak závislý. Překladače nových jazyků je možné implementovat pomocí jiných technologií.

Překladač vytvořený pomocí `PLY` se skládá ze dvou částí – lexikální a syntaktický analyzátor. Oba jsou implementovány jako samostatné třídy (lze je však implementovat na úrovni modulů). Lexikální analyzátor definuje množinu lexikálních jednotek, které jsou ve zdrojovém souboru rozpoznávány. Jsou definovány pomocí regulárních výrazů buď jako statické atributy třídy, nebo, je-li vyžadováno další zpracování, jako metody. V obou případech musí mít název atributu/metody prefix `t_`. Definice lexikální jednotky jako atributu třídy (detekce operátoru) má zápis:

```
t_LOGIC_AND = r'\&\&'
```

Definice lexikální jednotky jako metody třídy s dalším zpracováním (detekce identifikátorů a klíčových slov) má formu:

```
def t_ID(self, t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = self.reserved.get(t.value,'ID')
    return t
```

Pořadí definice lexikálních jednotek je důležité. Při procházení vstupního textu jsou nejdříve hledány ty lexikální jednotky, které jsou definovány jako funkce (v pořadí, ve kterém jsou definovány). Následně jsou hledány ty, které jsou definovány jako atributy. Zde je rozhodující délka regulárního výrazu – delší jsou zpracovány dříve.

Jednotlivé metody syntaktického analyzátoru (s prefixem `p_`) jsou brány jako gramatická pravidla. Samotná pravidla jsou zapsána opět jako dokumentační řetězec funkce v Backus–Naurově formě. Ukázka zápisu gramatického pravidla:

```
def p_expr_plus(self, p):
    'expr : LPAREN PLUS expr expr RPAREN'
    p[0] = p[3] + p[4]
```

Jednotlivé neterminály jsou dále zpracovány v definovaném pořadí. Jakmile jsou zpracovány všechny, je proveden kód samotné metody. Parametr metody `p` je kolekce hodnot všech terminálů a již expandovaných neterminálů, které se v tomto pravidle vyskytují. Jsou přístupné pomocí indexů kolekce `p`. Položka kolekce `p[0]` slouží jako výstup expandování tohoto pravidla.

Tímto způsobem je sestaven abstraktní syntaktický strom interního simulačního modelu. Jeho další transformace na simulační model již není v režii vstupního překladače, díky čemuž je transformace nezávislá na vstupní jazyce.

4.2 Grafické uživatelské rozhraní

Pro vizualizaci průběhu simulace poskytuje simulátor plnohodnotné uživatelské rozhraní. Je implementováno pomocí frameworku `PyQt5` [8], což je sada vazeb jazyka python na aplikační framework `Qt5` [7]. Framework je multiplatformní, což je vzhledem k požadavkům na přenositelnosti simulátoru důležitá vlastnost. Samotné vazby `PyQt5` jsou implementovány jako sada modulů jazyku python.

Uživatel má k dispozici grafický prvek, ve kterém jsou zobrazeny veškeré stavové prvky modelu. Ty může přidat do seznamu prvků, které si přeje vizualizovat (tzv. *workset*), který navíc uchovává informace o způsobu vykreslení. Mimo reference na instanci vykreslovaného prvku obsahuje informaci o barvě a datovém formátu, které si uživatel přeje při vykreslení aplikovat.

Vykreslování vlnových grafů řeší modul `dfsim.view.gui.pyqt5.plot`. Plocha pro vykreslování grafů je reprezentována třídou `PlotWidget`, která rozšiřuje třídu `QWidget` frameworku `PyQt5`. Jedná se tedy o standardní komponentu, kterou lze pomocí libovolného správce rozložení zapojit do grafického rozhraní.

Komponenta vykresluje vždy jen viditelnou část všech zobrazených vlnových grafů. Nejdříve se vypočítá rozsah viditelných časových jednotek. První viditelný okamžik je zjištěn z hodnoty posuvníku komponenty. Vizualizovány jsou hodnoty všech položek z *worksetu* v daném časovém rozsahu. Způsob vykreslení se liší podle toho, zda se jedná o jednobitový, nebo vícebitový prvek modelu.

Kapitola 5

Simulační experimenty a testování

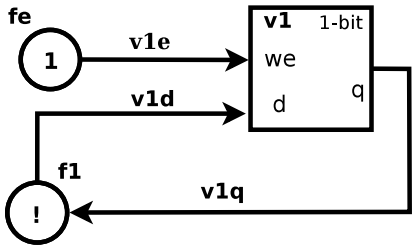
Během vývoje simulátoru byly průběžně prováděny simulační experimenty jak jednodušších, tak složitějších vstupních modelů. Po manuálním ověření výsledků byly tyto experimenty automatizovány. Nyní jejich výstupy slouží pro potřeby ověření správnosti chování simulátoru. Lze je tedy přirovnat k systémovým testům. Samostatná sekce je věnována sadě jednotkových testů jednotlivých komponent simulátoru.

5.1 Jednoduché simulační modely

Simulační experimenty s jednodušími modely mohou sloužit jak k ověření správnosti konkrétních algoritmů a interakce více modulů, jejichž testování na nižší úrovni by bylo komplikované, tak k ověření správnosti chování simulátoru jako celku.

5.1.1 Model invertoru

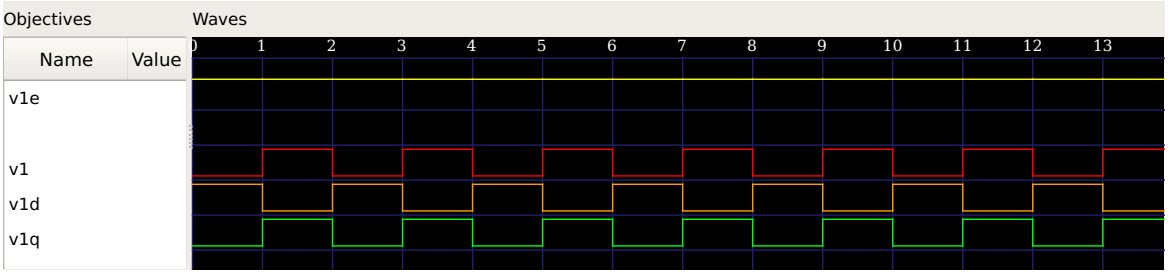
Model obsahuje jednobitový registr, jehož výstup je přes funkční uzel, který provádí negaci hodnot, zapojen zpět na vstup. Každý simulační krok tedy dochází k překlopení hodnoty registru z nuly na jedničku a naopak. Při vývoji simulačního jádra bylo cílem nejdříve dosáhnout úspěšné simulace právě tohoto modelu. Složitější konstrukce byly přidávány a testovány až následně.



Obrázek 5.1: Zapojení modelu 1-bitového invertoru.

Schéma zapojení je znázorněno na obrázku 5.1. Ačkoliv je podobné zapojení 4-bitového čítače (obrázek 3.2), na němž byla již dříve v této práci demonstrována řada algoritmů, cílem tohoto modelu je ověřit základní funkcionalitu.

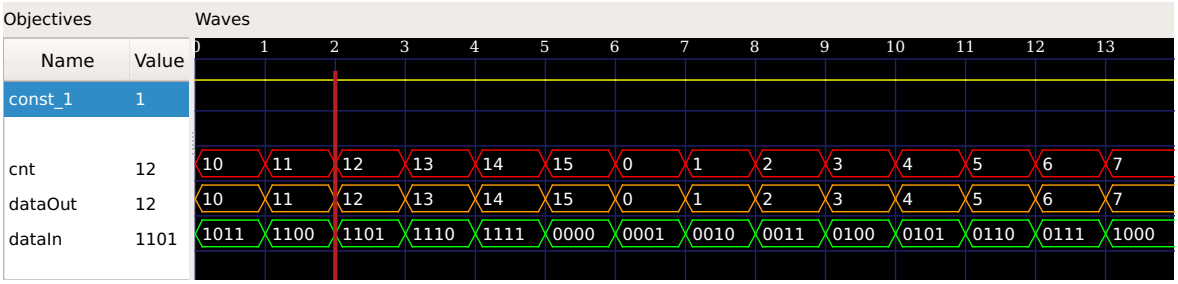
Obrázek 5.2 ukazuje průběh simulace. Je viditelné, že stav registru a hodnota výstupního signálu si jsou vždy rovny. Současně jsou opačné od hodnoty vstupu. V každém simulačním kroku pak dojde k záměně jejich hodnot.



Obrázek 5.2: Grafický výstup simulace modelu 1-bitového invertoru.

5.1.2 Model čtyřbitového čítače

Tento model byl již dříve v této práci využíván pro demonstraci použitých algoritmů a výstupních rozhraní. Schéma zapojení je znázorněno na obrázku 3.2. Obsahuje více-bitové prvky (registr i signály). Cílem experimentu je ověřit chování registru při rostoucí hodnotě. Čítač by měl po dosažení hodnoty $2^N - 1$, kde N je jeho bitová šířka, přetéct a začít počítat od nuly.



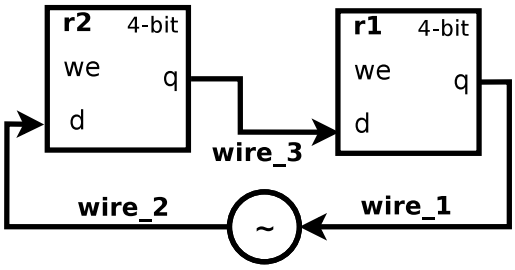
Obrázek 5.3: Grafický výstup simulace modelu 4-bitového čítače.

Součástí experimentu je soubor s počátečními stavy, po jehož aplikaci je registr inicializován na hodnotu 10. Průběh simulace je zobrazen na obrázku 5.3. Stav registru a hodnota jeho výstupního signálu je zobrazen v desítkové soustavě. Naopak datový vstup registru je v soustavě dvojkové. Vidíme, že po dosažení maximální hodnoty 4-bitového čítače dojde k jeho přetečení. Jelikož registry mají jednotkové zpoždění při zápisu, předbíhá hodnota vstupu hodnotu registru o jednu časovou jednotku.

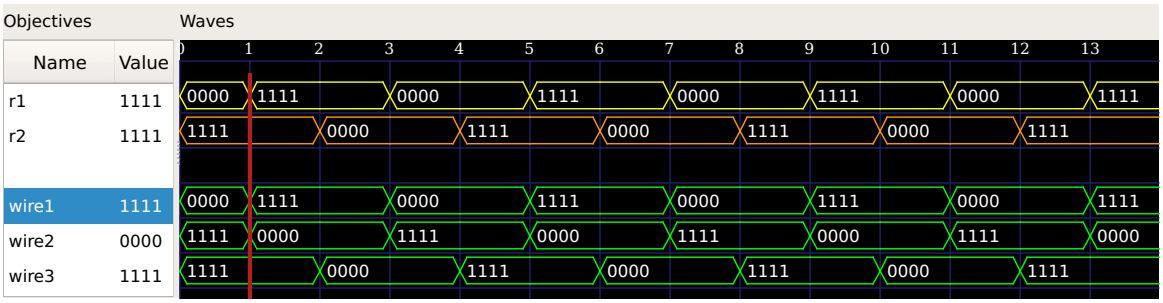
5.1.3 Model se zapojením více registrů za sebou

Tento experiment pracuje s modelem, který obsahuje dva registry zapojené do série. Touto technikou je možné modelovat zápisové zpoždění větší než jednotkové. Na počátku jsou registry inicializovány na binární hodnoty `0000` a `1111`. Ani jeden z registrů nemá k portu `we` připojen signál, zápis do registru je tedy vždy povolen. Schéma zapojení je ukázáno na obrázku 5.4.

Výstup simulace je ukázán na obrázku 5.5. Vidíme, že oba registry drží svou hodnotu po dvě časové jednotky. Další dvě časové jednotky pak registr má hodnotu s invertovanými bity.



Obrázek 5.4: Grafický výstup simulace modelu se sériovým zapojením registrů.



Obrázek 5.5: Grafický výstup simulace dvou registrů zapojených do série ve smyčce přes invertor.

5.2 Model 8-bitového procesoru

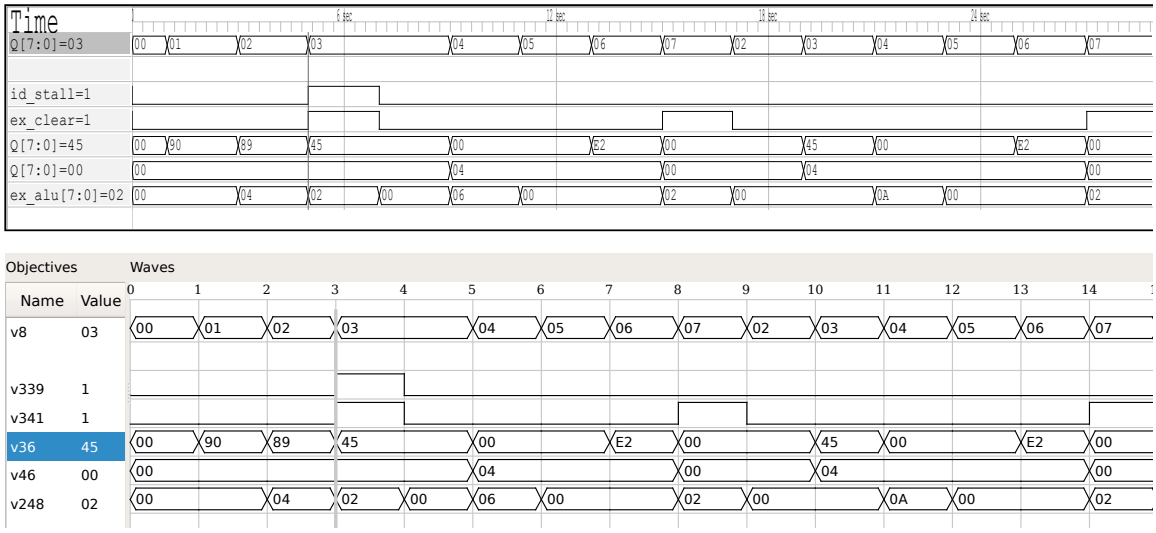
Tento model 8-bitového procesoru obsahuje tři fáze zřetězení. Byl vyvinut především pro testování nových verifikačních technik. Různé varianty (včetně zdrojových kódů VAM) jsou k nalezení na [13]. Zdrojové kódy jsou také součástí projektu na přiloženém CD. Simulační experimenty byly prováděny nad zjednodušenou variantou, ve které byly např. redukovány funkční uzly, které pouze přiřazují hodnoty dvou signálů. Pro ilustraci, zjednodušený model obsahuje 75 funkčních uzlů, 92 signálů, 14 registrů a 2 paměťové jednotky.

Existuje ekvivalentní model tohoto procesoru zapsaný v jazyce Verilog. Tato verze procesoru byla simulována pomocí simulátoru Icarus Verilog [24]. Porovnání simulace části modelů je ukázáno na obrázku 5.6. V horní části je výsledek simulace modelu pomocí simulátoru Icarus Verilog. Výsledky jsou vizualizovány pomocí nástroje GTKWave. V dolní části je ukázán průběh simulace modelu zapsaného ve VAM pomocí `dfsim`. Modely se liší pouze v názvech prvků, jinak se jedná o ekvivalentní modely, které produkují stejné výsledky.

5.3 Model 32-bitového procesoru

Byly provedeny také simulační experimenty komplexnějšího modelu 32-bitového procesoru s pěti fázemi zřetězení, který je schopen interpretovat podmnožinu instrukcí architektury DLX. Model je k nalezení na [13]. Zde jsme vzhledem ke složitosti modelu nebyli prozatím schopni ověřit správnost výsledků simulace. Přesto byly experimenty s tímto modelem užitečné.

Model obsahuje tři paměťové jednotky s šířkou adresy 32 bitů (neobsahují maximální počet buněk). Nicméně během simulačních experimentů jsme zjistili, že právě počet paměťových buněk může být pro simulaci kritický. Simulátor nezvládá alokaci a efektivní



Obrázek 5.6: Porovnání simulace modelu 8-bitového procesoru pomocí simulátoru Icarus Verilog (nahoře) a `dfsim` (dole).

správu 2^{32} buněk, a proto simulace selže. Problémem je spíše prostorová, než časová náročnost. Proto je plánována implementace nového mechanismu alokace a správy buněk paměťových jednotek. Jednou z možností, které se nabízejí, je implementace paměťových jednotek řídkým způsobem, tedy uchovávat pouze paměťové buňky, jejichž hodnoty se změnily od hodnoty počáteční.

5.4 Testování

Na úrovni systémových testů jsou automatizovány simulační experimenty vybraných modelů, k nimž je dodán očekávaný výstup, který se porovnává s výstupem simulátoru. Jednotkové testy se pak zaměřují na ověření správnosti chování jednotlivých komponent simulátoru na nižší úrovni.

Testovací sada obsahuje jednotkové testy pro:

- bitové vektory – jejich vytváření, transformace a operace nad nimi,
- model – především přidávání a vytváření jednotlivých prvků,
- jednotlivé prvky modelu – signál, registr, paměť, funkční uzel,
- evaluační plán – sestavení na základě existujícího modelu,
- kalendář událost – přidávání událostí, kontrola správného řazení, atp.

Kapitola 6

Závěr

Cílem této práce bylo vytvořit interaktivní simulátor pro grafy toku dat. Simulátor vzniká jako podpůrný nástroj pro formální verifikaci návrhů mikroprocesorů v rámci projektu HADES. Jeho úkolem je snížení potřebný čas a zvýšit kvalitu procesu verifikace.

Vstupem simulátoru je graf toku dat popsaný v jazyce VAM, který slouží k popisu modelů mikroprocesorů na úrovni meziregistrových přenosů, na jehož základě je vytvořen interní simulační model. Aby bylo možné provádět simulaci nad tímto modelem efektivně, byla zavedena řada optimalizací a specifických algoritmů, které eliminují nadbytečná vyhodnocení. Pro simulační model je sestaven evaluační plán, který určuje pořadí vyhodnocování prvků. Jsou-li prvky vyhodnocovány v tomto pořadí, pak je každý uzel v jednom simulačním kroku vyhodnocen právě jednou. Byl také navržen speciální simulační algoritmus, který za běhu provádí další eliminaci nadbytečných vyhodnocení. Simulační jádro pracuje jako interpret uživatelských příkazů. Podporovány jsou příkazy pro řízení simulace a manipulaci se stavem interního modelu. Uživatel tedy není omezen přímým výstupem simulace, ale běh může přizpůsobit svým potřebám.

Simulátor poskytuje uživateli několik přístupů k analýze simulace. První poskytuje přímý textový výstup simulace ve formátu VCD, díky čemuž je možné použít výsledky simulace v dalších nástrojích. K němu existuje také interaktivní varianta, která umožňuje uživateli zadávat příkazy. Toto rozhraní je určeno pro vytváření automatizovaných testů. Uživatel může vytvořit sekvenční program sestávající se z příkazů simulátoru a předat ho simulátoru na vstup, který jej interpretuje. K dispozici je také plnohodnotné grafické rozhraní, které vizualizuje průběh simulace formou digitálních vlnových grafů. Tím, že má uživatel možnost průběh simulace vidět, je schopen snadněji pochopit chování modelu.

Byly provedeny simulační experimenty jednodušších i složitějších modelů. Ty jednodušší slouží především k ověření správnosti chování specifických částí modelu a simulačního jádra. Byly proto zařazeny do testovací sady simulátoru. Vedle toho byla provedena série experimentů s modelem 8-bitového procesoru, kde bylo dosaženo stejných výsledků simulace jako široce používaný simulátor Icarus Verilog.

V budoucnu je plánováno postupné rozšiřování sady uživatelských příkazů o nové užitečné nástroje. Vedle toho byly lokalizovány určité části simulátoru, kde by bylo vhodné provést optimalizace. Nejzajímavěji se však jeví koncept podmíněné simulace. Simulátor by prováděl simulační kroky do doby, než nastane nějaká událost nebo začne platit určitá podmínka (např. hodnota dvou registrů se rovná). Tento mechanismus by výrazně zvýšil schopnost simulátoru automaticky detekovat chyby v chování modelu.

Literatura

[1] The Value Change Dump File. http://support.ema-eda.com/search/eslfiles/default/main/sl_legacy_releaseinfo/staging/sl3/release_info/psd142/vlogref/chap20.html, 1999, [cit. 2015-05-09].

[2] GHDL Features. http://home.gna.org/ghdl/features.html, 2005, [cit. 2015-01-20].

[3] GHDL Homepage. http://home.gna.org/ghdl/, 2005, [cit. 2015-01-20].

[4] GHDL Manual. http://home.gna.org/ghdl/manual.html, 2005, [cit. 2015-01-20].

[5] GTKWave 3.3 Wave Analyzer User's Guide. http://gtkwave.sourceforge.net/gtkwave.pdf, 2014, [cit. 2014-01-20].

[6] Mentor: ModelSim ASIC and FPGA Design. http://www.mentor.com/products/fv/modelsim/, 2015, [cit. 2015-01-20].

[7] The Qt Company: Qt — Cross-platform application & UI development framework. http://www.qt.io/, 2015, [cit. 2015-05-03].

[8] Riverbank: What is PyQt? http://www.riverbankcomputing.co.uk/software/pyqt/intro, 2015, [cit. 2015-05-03].

[9] Beazley, D. M.: PLY (Python Lex-Yacc). http://www.dabeaz.com/ply/, [cit. 2015-05-03].

[10] Bernard, B.: Prezentační vzory z rodiny MVC [online]. 2009, [cit. 2015-05-10].

[11] Bhattacharyya, S. S.; Murthy, P. K.; Lee, E. A.: *Software Synthesis from Dataflow Graphs*. Springer, 1996, ISBN 0-7923-9722-3.

[12] Charvát, L.; Smrčka, A.; Vojnar, T.: Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors. In *Proceedings of 15th International Workshop on Microprocessor Test and Verification (MTV 2014)*, IEEE Computer Society, 2014, ISBN 978-1-4673-6858-2, s. 83–89.

[13] Charvát, L.; Smrčka, A.; Vojnar, T.: Hades – Nástroj pro verifikaci hardware [online]. http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/, 2015 [cit. 2015-02-15].

[14] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, první vydání, 1994, ISBN 0201633612.

[15] Johnson, S. C.: Yacc: Yet Another Compiler-Compiler. http://dinosaur.compilertools.net/yacc/, [cit. 2015-05-03].

[16] Lee, E. A.; Messerschmitt, D. G.: Synchronous Data Flow. In *Proceedings of the IEEE*, ročník 75, 1987, s. 1235–1245.

[17] Lesk, M. E.; Schmidt, E.: Lex – A Lexical Analyzer Generator. http://dinosaur.compilertools.net/lex/, [cit. 2015-05-03].

[18] Mealy, B.; Tappero, F.: Free Range VHDL [online]. http://www.freerangefactory.org/dl/free_range_vhdl.pdf, 2013, [cit. 2015-01-25].

[19] Priya, M. M.: Topological Sorting. http://www.cs.iit.edu/~cs560/fall_2012/Research_Paper_Topological_sorting/Topological%20sorting.pdf, [cit. 2015-04-03].

[20] Schaumont, P.: *A Practical Introduction to Hardware/Software Codesign*, kapitola Data Flow Modeling and Transformation. Springer, druhé vydání, 2013, ISBN 978-1-4614-3737-6.

[21] Smrčka, A.: Variable-Assignment-Model (VAM) – Structure and Language [online]. http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/download/language.txt, 2012, [cit. 2015-01-17].

[22] Wikipedia: Verilog [online]. http://en.wikipedia.org/wiki/Verilog, 2014, [cit. 2015-01-25].

[23] Wikipedia: And-inverter graph [online]. http://en.wikipedia.org/wiki/And-inverter_graph, 2014 [cit. 2015-01-20].

[24] Williams, S.: Icarus Verilog. http://iverilog.icarus.com/, 2015, [cit. 2015-05-02].

Příloha A

Spouštění simulátoru

```
dfsim.py  [-h] [--init INIT] [-r RUN] [-l {vam}] [-q] [--text | --itext | --gui]
          [-o OUTFILE] [infile]
```

Poziční argumenty:

`infile`	vstupní soubor (volitelný pouze v grafickém režimu)

Volitelné argumenty:

`-h, --help`	vytiskne nápovědu programu
`--init INITFILE`	jméno souboru obsahující konfiguraci počátečního stavu modelu
`-r RUN, --run RUN`	počet simulačních kroků, které se po spuštění provedou (implicitně 0)
`-q, --quiet`	eliminuje tisk některých varovných hlášek
`-l {vam}, --lang {vam}`	nastaví vstupní jazyk (implicitně VAM – zatím jediný podporovaný)
`--text`	přepne simulátor do textového výstupního režimu
`--itext`	přepne simulátor do interaktivního textového výstupního režimu
`--gui`	přepne simulátor do grafického výstupního režimu
`-o OUTFILE, --output OUTFILE`	výstupní soubor, použití se liší podle výstupního režimu: textový – VCD výstup simulace, interaktivní textový – výstup uživatelských příkazů, grafický – nemá výstup.

Příloha B

Obsah CD

```
/
├── demo                    ... demonstrační nástroje z konference Excel@FIT
│   ├── demo.mp4            ... video
│   └── poster.pdf          ... plakát
├── dfsim/                  ... adresář projektu
│   ├── dfsim/              ... modul první úrovně zdrojových kódů
│   ├── examples/           ... zdrojové kódy simulačních modelů
│   ├── tests/              ... automatizované testy a simulační experimenty
│   ├── res/                ... obrázky a podobné zdroje
│   ├── dfsim.py            ... hlavní skript simulátoru
│   └── README.md
├── docs
│   ├── tex/                ... zdrojové kódy textu práce
│   ├── paper.pdf           ... článek prezentovaný na konverenci Excel@FIT
│   ├── thesis.pdf          ... text práce s aktivními odkazy
│   └── thesis-print.pdf    ... text práce určený pro tisk
└── lib                     ... knihovny třetích stran v patřičné verzi
    ├── ply-3.6.tar.gz
    └── PyQt-gpl-5.4.1.tar.gz
```